Machine learning techniques for automatic knowledge graph completion

Armand Boschin

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAT016

Thèse de doctorat

Machine learning techniques for automatic knowledge graph completion

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Mathématiques et Informatique

Thèse présentée et soutenue à Palaiseau, le 21 avril 2023, par

ARMAND BOSCHIN

Composition du Jury :

Amel Bouzeghoub Professeure, Télécom SudParis	Présidente/examinatrice
Sébastien Ferré Professeur, Université de Rennes - CNRS - IRISA	Rapporteur
Nathalie Pernelle Professeure, Université Sorbonne Paris Nord - LIPN	Rapporteur
Luis Galárraga Chercheur permanent, INRIA - IRISA	Examinateur
Thomas Bonald Professeur, Télécom Paris	Directeur de thèse

Résumé

Un graphe de connaissances est un graphe orienté dont les nœuds sont des entités et les arêtes, typées par une relation, représentent des faits connus liant les entités. Ces graphes sont capables d'encoder une grande variété d'information mais leur construction et leur exploitation peut se révéler complexe. Historiquement, des méthodes symboliques ont permis d'extraire des règles d'interaction entre entités et relations, afin de corriger des anomalies ou de prédire des faits manquants. Plus récemment, des méthodes d'apprentissage de représentations vectorielles, ou plongements, ont tenté de résoudre ces mêmes tâches. Initialement purement algébriques ou géométriques, ces méthodes se sont complexifiées avec les réseaux de neurones profonds et ont parfois été combinées à des techniques symboliques antérieures. Cette thèse est constituée de six chapitres structurés de la manière suivante.

Le premier chapitre commence par retracer rapidement l'histoire des bases de connaissances depuis les projets historiques jusqu'à Wikidata. Est ensuite introduite la tâche de complétion de graphe de connaissances ainsi que trois de ses sous-tâches habituelles, la prédiction de liens, la prédiction de relations et le typage d'entité.

Le deuxième chapitre introduit les définitions formelles nécessaires à la construction d'une base de connaissance et d'une ontologie, c'est-à-dire à la constitution de faits accompagnés d'une taxonomie et d'axiomes logiques de construction. La notion de graphe est ensuite introduite de manière à décrire formellement la structure formée par les faits constitutifs des bases de connaissances.

Le troisième chapitre propose ensuite un état des lieux des techniques existantes pour la complétion automatique de graphe de connaissances. Pour commencer, une définition de l'apprentissage automatique est donnée à l'aide du formalisme des fonctions paramétriques dont les paramètres sont estimés sur des données d'entrainement. Cela sert à introduire les modèles de plongement et plus particulièrement ceux s'appliquant aux graphes de connaissances qui sont ensuite présentés en détail. Quelques méthodes combinatoires reposant principalement sur des règles d'associations sont présentées.

Le quatrième chapitre s'intéresse au problème de l'implémentation. En effet, la grande diversité des bibliothèques utilisées rend difficile la comparaison des résultats obtenus par différents modèles. Dans ce contexte, la bibliothèque Python TorchKGE a été développée afin de proposer un environnement unique pour l'implémentation de modèles de plongement et un module hautement efficace d'évaluation par prédiction de liens. Cette bibliothèque repose sur l'accélération graphique de calculs tensoriels proposée par PyTorch, est compatible avec les bibliothèques d'optimisation usuelles et est disponible en source ouverte. Son adoption croissante pas la communauté montre son intérêt.

Le cinquième chapitre porte sur l'enrichissement automatique de Wikidata par typage des hyperliens liant les articles de Wikipedia. Une étude préliminaire a montré que le graphe des articles de Wikipedia est beaucoup plus dense que le graphe de connaissances correspondant dans Wikidata. Une nouvelle méthode d'entrainement impliquant les relations et une méthode d'inférence utilisant les types des entités ont été proposées

et des expériences ont montré la pertinence de l'approche. Un nouveau jeu de données, WDV5 est également proposé.

Enfin, le sixième chapitre propose aborde le typage automatique d'entités comme une tâche de classification hiérarchique. L'étude de cette tâche a mené à la conception d'une fonction d'erreur hiérarchique, utilisable pour l'entrainement de modèles tensoriels, ainsi qu'un nouveau type d'encodeur. Des expériences ont permis une bonne compréhension de l'impact que peut avoir une connaissance a priori de la taxonomie des classes sur la classification. Elles ont aussi renforcé l'intuition que la hiérarchie peut être apprise à partir des données si le jeu est suffisamment riche.

Abstract

A knowledge graph is a directed graph in which nodes are entities and edges, typed by a relation, represent known facts linking two entities. These graphs can encode a wide variety of information, but their construction and exploitation can be complex. Historically, symbolic methods have been used to extract rules about entities and relations, to correct anomalies or to predict missing facts. More recently, techniques of representation learning, or embeddings, have attempted to solve these same tasks. Initially purely algebraic or geometric, these methods have become more complex with deep neural networks and have sometimes been combined with pre-existing symbolic techniques.

In this thesis, we first focus on the problem of implementation. Indeed, the diversity of libraries used makes the comparison of results obtained by different models a complex task. In this context, the Python library TorchKGE was developed to provide a unique setup for the implementation of embedding models and a highly efficient inference evaluation module. This library relies on graphic acceleration of tensor computation provided by PyTorch, is compatible with widespread optimization libraries and is available as open source.

We then consider the automatic enrichment of Wikidata by typing the hyperlinks linking Wikipedia pages. A preliminary study showed that the graph of Wikipedia articles is much denser than the corresponding knowledge graph in Wikidata. A new training method involving relations and an inference method using entity types were proposed and experiments showed the relevance of the combined approach, including on a new dataset.

Finally, we explore automatic entity typing as a hierarchical classification task. That led to the design of a new hierarchical loss used to train tensor-based models along with a new type of encoder. Experiments on two datasets have allowed a good understanding of the impact a prior knowledge of class taxonomy can have on a classifier but also reinforced the intuition that the hierarchy can be learned from the features if the dataset is large enough.

List of Publications

WikiDataSets: Standardized sub-graphs from Wikidata.
Boschin, A., Bonald, T. (2019)
Preprint.
Reference [19]

Knowledge Representation and Rule Mining in Entity-Centric Knowledge Bases.
Suchanek, F., Lajus, J., Boschin, A., Weikhum, G. (2019)
Postprint presented at the *15th Reasoning Web Summer School* (Bolzano, Italy, September 2019).
Reference [116]
Personal contribution: Section 4 (Representation Learning)

TorchKGE: Knowledge Graph Embedding in Python and PyTorch
Boschin, A. (2020)
Postprint presented at the *International Workshop on Knowledge Graph co-located with the 26th Conference on Knowledge Discovery and Data Mining* (August 2020).
Reference [18]

Enriching Wikidata with Semantified Wikipedia Hyperlinks.
Boschin, A., Bonald, T. (2021)
Postprint presented at the *2nd Wikidata Workshop co-located with the 20th International Semantic Web Conference* (October 2021).
Reference [20]

Combining Embeddings and Rules for Fact Prediction.
Boschin, A., Jain, N., Keretchashvili,G., Suchanek,F. (2022)
Postprint presented at the *International Research School in Artificial Intelligence* (Bergen, Sweden, June 2022).
Reference [22]
Personal contribution: parts of Sections 3 (Embedding models), parts of Section 4 (Embedding Methods with Logical Components) and Section 5 (Rule Mining with embedding techniques).

A Self-Encoder for Learning Nearest Neighbors.
Boschin, A., Bonald, T., Jeanmougin, M. (2023)
Under review at ECML23
Reference [21]

Contents

Notations

- Given $\mathbb{S}$ a set with group structure, $\mathbb{S}^*$ is the same set without its identity element.
- $\mathbb{N}$ is the set of natural numbers. $\mathbb{N}^*$ is the set of strictly positive natural numbers.
- $\mathbb{K}$ denotes a field (usually either $\mathbb{R}$ or $\mathbb{C}$).
- $\mathbb{R}$ (resp. $\mathbb{C}$) is the field of real (resp. complex) numbers.
- Given a set A, $|A| \in \mathbb{N}$ is the cardinal of A.
- Given a set A, $\mathcal{P}(A)$ denotes the set of all subsets of A.
- Vectors are denoted with bold letters.
- Any vector $\boldsymbol{v} \in \mathbb{K}^k$ can indifferently be considered as a matrix of $\mathbb{K}^{k\times 1}$.
- Given any matrix $M \in \mathbb{K}^{n\times m}$, $M^T \in \mathbb{K}^{m\times n}$ is the transposition of M.
- Given a complex number $x \in \mathbb{C}$, $\bar{x}$ is the complex conjugate of x. For a vector $\boldsymbol{x} \in \mathbb{C}^n$, $\bar{\boldsymbol{x}}$ is a complex vector such that for any $i \in \{1, 2, \ldots, n\}$, $(\bar{\boldsymbol{x}})_i = \overline{(\boldsymbol{x}_i)}$.
- Re and Im designate the real and imaginary operators such that for any $x \in \mathbb{C}$, $x = \mathrm{Re}(x) + i\,\mathrm{Im}(x)$ with $(\mathrm{Re}(x), \mathrm{Im}(x)) \in \mathbb{R}^2$. Those are naturally extended to complex vectors.
- Given two matrices $(A, B) \in \mathbb{K}^{n\times k} \times \mathbb{K}^{k\times m}$, $A \cdot B \in \mathbb{K}^{n\times m}$ is the matrix product of A and B.
- I_k is the k-dimensional identity matrix and $\mathbb{1}_k$ is the all-one vertical vector of $\mathbb{R}^k$. The dimension k of these will be omitted except when ambiguous.
- Given a vector $\boldsymbol{x} \in \mathbb{K}^n$, $\mathrm{diag}(\boldsymbol{x}) \in \mathbb{K}^{n\times n}$ is the diagonal matrix with the elements of $\boldsymbol{x}$ on the diagonal.
- $\Longrightarrow$ denotes the logical implication and $\Longleftrightarrow$ denotes the logical equivalence.

Acronyms

API	Application programming interface
BalNS	Balanced negative sampling
BCE	Binary cross-entropy
BerNS	Bernoulli negative sampling
CD	Continuous development
CI	Continuous integration
CWA	Closed world assumption
DAG	Directed acyclic graph
DVCS	Distributed version control system
FN	False negative
FP	False positive
GCN	Graph convolutional networks
GPU	Graphics processing unit
HC	Hierarchical classification
ILP	Inductive logic programming
KB	Knowledge base
KG	Knowledge graph
LCN	Local classifier per node
LCPN	Local classifier per parent node
LCWA	Local closed world assumption
LLM	Large language model
ML	Machine learning
MLP	Multi-layer perceptron
MR	Mean rank
MRR	Mean reciprocal rank
NN	Nearest neighbors
NS	Negative sampling
OWA	Open world assumption
OWL	Web ontology language
PCA	Partial completeness assumption
RDF	Resource description framework
RDFS	Resource description framework schema
RL	Representation learning
SVD	Singular value decomposition
SVM	Support vector machine
TF	Type filtering
TN	True negative
TorchKGE	Knowledge graph embedding in PyTorch
TP	True positive
URI	Uniform resource identifier
URL	Uniform resource locator

Chapter 1

Introduction

1.1 Context of the thesis

One of the most ambitious projects of knowledge collection in history is the famous Great Library of Alexandria. Established in the third century before Christ's birth, it quickly expanded notably by copying all books found on the numerous ships that docked in the Alexandria harbor and by keeping the originals while returning the copies to the owners. This famous enterprise is just an example of the numerous attempts of mankind to centralize knowledge. Successive societies came up with various solutions to the problem formalized in the eighteenth century by Denis Diderot in its Encyclopedia as: *"collect[ing] knowledge disseminated around the globe; to set forth its general system to the men with whom we live and transmit it to those who will come after us"* [37].

Diderot's ambitious project found a modern embodiment with the internet and the birth of Wikipedia: a global, open-source and collaborative project. Jimmy Wales, a co-founder of Wikipedia presented it in the following terms: *"Imagine a world in which every single person on the planet is given free access to the sum of all human knowledge. That's what we are doing."*. First launched in 2001, Wikipedia has become an unavoidable tool of the modern era. It is widely use by people around the world to document themselves on a large variety of subjects. It currently counts nearly sixty million articles in more than three hundred languages. The format of textual articles, however, shows some limitations for some modern usage. Particularly, textual representation of natural language is not easily queryable for machines and algorithms, thus limiting its processing by modern *intelligent systems* such as recommender systems or chatbots.

In the last decades, the concept of knowledge base has come to designate databases designed to record and organize knowledge mainly by listing entities and verified facts involving these entities. There is a large variety of knowledge bases. Some are public and about general knowledge such as ConceptNet, Freebase and Wikidata. Such projects are usually open-source and rely on public contributions to grow. Others are topic-specific such as WordNet, which specifically records knowledge in the field of linguistics. All the public knowledge bases available online form what is called the semantic web. On the other hand, many knowledge bases are not public and are simply projects that private companies launch to organize their domain expertise, as an operational necessity or for archiving purposes. Eventually, larger private knowledge bases can be used by tech companies to propose services to the public, such as personal assistants.

In term of public knowledge bases, the most unavoidable project is certainly Wikidata. In 2012, the Wikimedia foundation, parent organization of Wikipedia, launched a new project to organize the knowledge contained in textual articles in the form of a structured knowledge base that would be easily queryable and processable by machines. Wikidata was born and would soon become the largest public knowledge base [127].

By essence, knowledge bases aim at growing as large as possible to become complete. Because of their size, they can quickly become difficult to process manually. This is the reason why methods to automatize processing tasks have been developed simultaneously with the knowledge bases themselves. Examples of tasks that researchers have continuously tried to automatize are the completion by fact suggestion and the correction by erroneous facts search. The first proposed methods were symbolic and rely on combinatorics to mine patterns in the bases. Those patterns can either be used to raise unusual facts as possible errors or to propose missing facts. Despite being historically the first, symbolic methods still prove to solve efficiently some problems.

In the same way as combinatorics, a historically prominent field in data pattern mining is statistics. Statistical theory indeed provides a lot of tools to estimate underlying distributions of observed data. More recently, the field of machine learning, which originally used mainly statistical methods, saw a large amount of new techniques emerge with the drop in the cost of computing power. A tipping point was the recent accessibility of graphics processing units over the past fifteen years that democratized the development and the use of models requiring heavy tensor computations. One of the consequences was the revolutionary performances of neural networks in the field of computer vision. The field of knowledge bases also benefited from this cheap access to computation power and many new machine learning models were proposed to automatically solve tasks such as construction, completion and correction by involving vector representations of knowledge bases.

1.2 Thesis outline

The problem tackled in this thesis is the automatic completion of knowledge graphs using machine learning techniques. A requirement to any pertinent contribution, is a proper introduction of the concepts at hand. Chapter 2 introduces the building blocks of knowledge bases and taxonomies along with the notion of graph, thus introducing knowledge graphs. In its simplest form, a knowledge graph is a collection of facts in the form of triples $\langle h, r, t \rangle$ were a relation r links a head entity h to a tail entity t. An example of a fact can be $\langle Paris, capital\ of, France \rangle$. Completing a knowledge graph then comes down to finding new triples that are likely to be true given the existing ones. A toy example is shown in Figure 1.1.

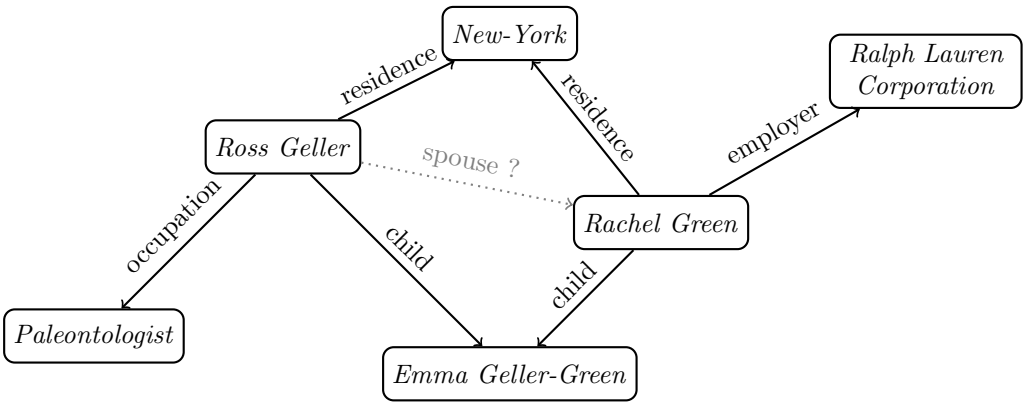


Figure 1.1: Toy example of a knowledge graph. The fact in gray, involving Ross Geller and Rachel Green, is unknown but it could be automatically suggested as likely, knowing that they are parents of the same child and live in the same city.

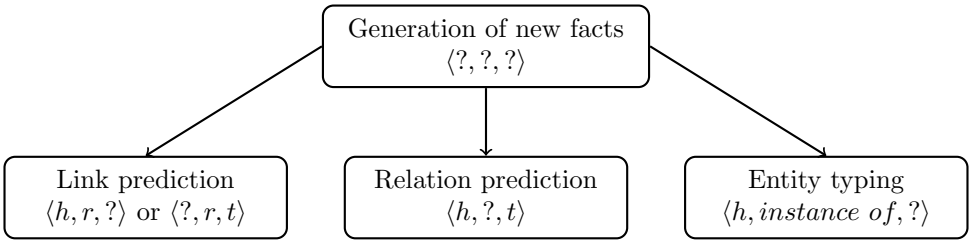


Figure 1.2: General task of knowledge graph completion and three restricted versions of it. h, r and t respectively designate the head entity, tail entity and relation and "?" is the missing part of the fact.

Following these definitions, Chapter 3 gives a proper framework to the task of automatic knowledge graph completion and then reviews the existing methods. The focus of the chapter is on machine learning techniques mainly relying on vector representations, or embeddings, of entities and relations. Symbolic methods are also introduced as they help put the performances of the machine learning techniques in perspective and are useful to design hybrid methods. This chapter particularly highlights the challenge of assessing the performance of various models in a comparable setting as well as the potential of simple models to perform comparably to over-engineered ones.

A first contribution of this thesis is then presented in Chapter 4: TorchKGE for Knowledge Graph Embedding in PyTorch. This is a Python library that provides a unique setting in which machine learning models for knowledge graph completion can be easily implemented and experimented with. The design of TorchKGE makes it compatible with the use of other widespread machine learning libraries, while providing tools specific to knowledge graphs. A key feature is a very efficient module to evaluate models on inference, a key task when tuning the parameters of a model.

Most of the introduced methods mine patterns from existing facts of the knowledge graph to be able to assess the likelihood of unknown ones. A naive way to apply those would be to evaluate all the possible combinations of two entities and a relation to select the most likely ones. The number of candidate triples is however very large and applying possibly heavy models to that task would be costly. A reasonable solution is to use alternative data sources to reduce the set of possible candidates in an informed manner. The resulting tasks that are tackled in the thesis are schematically represented in Figure 1.2.

There are several possible sources of candidates and a good example is Wikipedia and the natural graph structure formed by the pages and the hyperlinks between them. If the pages of two entities are linked, it might indicate a semantic relation between the two and subsequently a possible fact. For example, the Wikipedia page of *Ross Geller* is linked to the one of *Rachel Green* by a link embedded in the sentence: "*His romantic feelings towards Rachel Green are an ongoing theme of his narrative arc.*"

Chapter 5 proposes a method to enrich the Wikidata knowledge graph by predicting the semantic meaning of hyperlinks existing between Wikipedia pages. The two entities of the fact-to-be come from the hyperlink and only the relation is missing. The resulting task is called relation-prediction. The method mainly relies on simple embedding models trained on the facts already existing in the knowledge graph. Three contributions are presented in this chapter: first a new training method improving the performance of models when specifically predicting missing relations, second, an inference protocol exploiting entity types with a simple and yet efficient symbolic technique and eventually, a new dataset extracted from Wikidata and called WDV5.

Another source of candidates simply comes from the typing relation that exists in knowledge bases and that characterizes entities with types hierarchically organized in a taxonomy. Figure 1.3 presents a small taxonomy involving types that could apply to en-

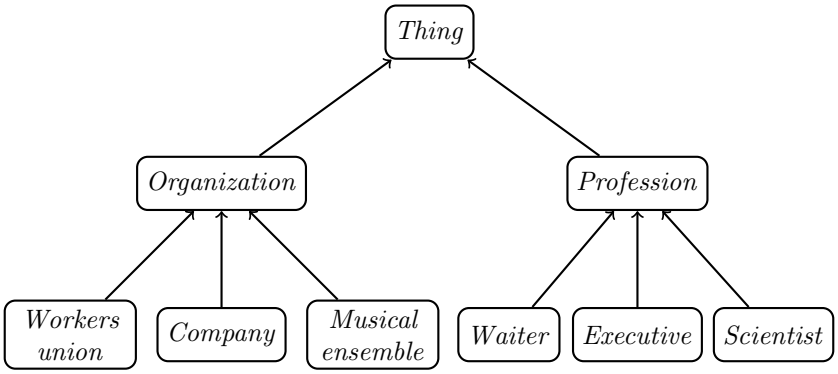


Figure 1.3: Example of a possible class taxonomy.

tities of the toy knowledge graph of Figure 1.1. If the taxonomy is rich enough, any entity could be typed. The resulting candidates are of the form $\langle entity, instance\ of, ? \rangle$. For example, some possible new facts involving the types of the taxonomy in Figure 1.3 are $\langle Ralph\ Lauren, instance\ of, Company \rangle$ and $\langle Paleontologist, instance\ of, Scientist \rangle$. Entity typing is a key task of knowledge graph completion because types are likely to give a very clear description of each entity. They are also making it possible to constrain facts, resulting in a well-constructed and coherent knowledge graph.

Motivated by entity typing, Chapter 6 dives into the problem of hierarchical classification by exploring various ways to include a known taxonomy in a machine learning classifier. A new hierarchical loss function is proposed to train tensor-based models and a simple, yet powerful, unsupervised Self-Encoder model is introduced. While the first improves the hierarchical compliance of predictions, the latter empirically gives an intuition of how much of the taxonomy is encoded in the knowledge graph structure.

Eventually a conclusive chapter puts the contributions of this thesis in perspective with the rest of the field to isolate promising axis for future works.

A schematic structure of the thesis is displayed in Figure 1.4.

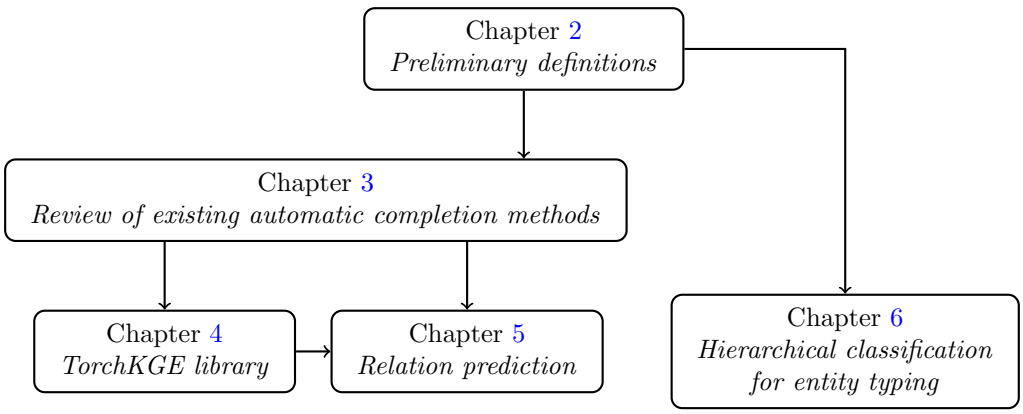


Figure 1.4: Structure of the thesis.

Chapter 2

Representing knowledge

The current chapter gives a comprehensive definition of knowledge bases and introduces the concept of knowledge graph.

Section 2.1 covers parts of Section 2 of the following article:

Knowledge Representation and Rule Mining in Entity-Centric Knowledge Bases.
Suchanek, F., Lajus, J., Boschin, A., Weikhum, G. (2019)
Postprint presented at the *15th Reasoning Web Summer School* (Bolzano, Italy, September 2019).
Reference [116]
Personal contribution: Section 4 (Representation Learning)

2.1 Entity-centric knowledge bases

There are three necessary building blocks in a Knowledge Base (KB): entities, relations and types. Those will be defined before formally introducing KBs.

2.1.1 Entities

Definition 1 (Entity)**.** *An entity is whatever may be an object of thought.*

It can be any conceivable thing: concrete or abstract, conceptual or material. For example, the concept of city is an entity as well as all the existing cities in the world. Referring to a specific entity can be tricky. Indeed, most entities have names (e.g. humans, places, events) but there is a large multiplicity in names. The city of New-York can either be designated by *New-York* or by its nickname *The Big Apple*. Conversely, *Paris* can either designate the city in France or the city in Texas or even a hero of Greek mythology. This calls for a mean to identify entities uniquely.

Definition 2 (Identifier)**.** *An identifier for an entity is a string of characters that represents the entity in a computer system.*

These identifiers can be chosen to be abstract (not human-intelligible). For example, Wikidata refers to entities by identifiers formed by an integer preceded by the letter *Q* (e.g. the *Freddie Mercury* entity's identifier in Wikidata is *Q15869*). These abstract identifiers are thought to be language-independent and stable in time. Even if another Freddie Mercury becomes a famous singer in the future, *Q15869* will still refer to the original one. It is usually assumed that there is a bijection between the set of entities of a

knowledge base and the set of identifiers and they are usually not distinguished, they are basically the entities. It is however necessary to have an intelligible and human-readable way to identify entities. This is done using labels.

Definition 3 (Label)**.** *A label for an entity is a human-readable string that names the entity.*

An entity can have several labels (they are synonymous) and a label can refer to several entities (it is then polysemous). However, not all entities have labels. Most chairs for example don't have particular labels, though some have: *Q4267023* is a throne in Tehran called the Marble Throne. Entities with labels are called *named-entities*. Though most KBs tend to record mostly named ones, unnamed-entities represent the vast majority of entities [98].

2.1.2 Entity types

The vast amount of entities of the world can be logically grouped together following some common features, precisely they can be typed.

Definition 4 (Type)**.** *Given a set of entities $\mathcal{E}$, a type (also: concept, class) $T \in \mathcal{P}(\mathcal{E})$ is a named set of entities that share a common trait. An entity of that set is called an instance of the type, and is said to have the type or belong to the type.*

Under this definition, the following are types of entities: the type of singers (i.e. the set of all people who sing professionally) but also the type of cities in France. Some instances of these types are, respectively, Freddie Mercury and Paris. Since everything is an entity, a type is also an entity and has an identifier and a label. Obviously, some types are related to one another and it is possible to define an order on types using the subsumption relation on types.

Definition 5 (Subsumption)**.** *Type A is a sub-type of type B if entities of type A are contained in the set of entities of type B, i.e. $A \subset B$. B is called a super-type of A.*

This binary relation is a partial order on the set of types of a KB. It is indeed obviously reflexive, anti-symmetric and transitive. It is not a total order, however, as it is not possible to order any arbitrary pair of types (for example two disjoint types can not be ordered). It is important not to confuse type inclusion with the relation between parts and wholes which can exist in a KB. For example, an arm is a part of the human body by the entity *arm* is not a sub-type of the entity *body*.

Given A and B two types, suppose A is a sub-type of B. A is said to be a *proper sub-type* of B if B contains more entities than A and A is a *direct sub-type* of B, if there is no type in the KB that is a super-type of A and a sub-type of B. Usually, when referring to sub-types, only direct sub-types are meant.

The partial order on types yields a natural hierarchy on the types: a taxonomy. The concept of graph, which is used in the following definition of taxonomy, will be properly defined in Section 2.3.1.

Definition 6 (Taxonomy)**.** *A taxonomy is a directed graph, where the nodes are types and there is an edge from type X to type Y if X is a proper direct sub-type of Y.*

The notion of taxonomy comes from biology. Zoological or botanic species form a taxonomy: *tiger* is a sub-type of *cat*, *cat* is a sub-type of *mammal*, and so on. This principle carries over to all other types of entities. For example, *internetCompany* is a sub-type of *company*, and *company* is a sub-type of *organization*. Since a taxonomy models proper inclusion, it follows that the taxonomic graph is acyclic: if a type is the sub-type of another type, then the latter cannot be a sub-type of the former. Thus, a taxonomy is a directed acyclic graph.

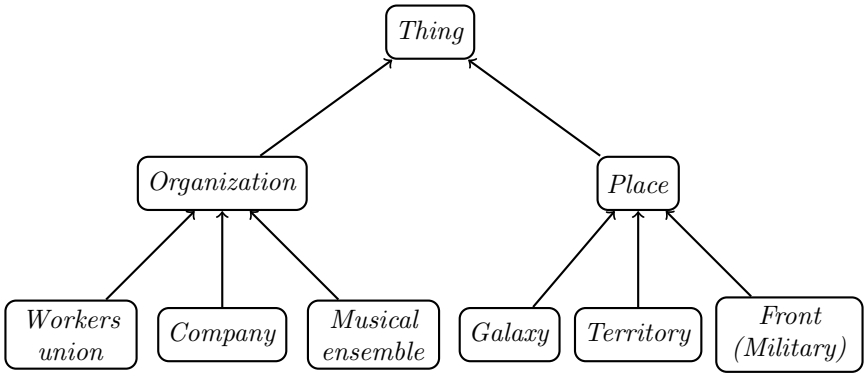


Figure 2.1: Extract of a taxonomy.

In some taxonomies, nodes have at most one out-going edge, turning it into a tree. In other cases, types can have several super-types. For example in Wikidata, the type *military officer* is a sub-type of both types *military personnel* and *officer*.

Usually, taxonomies are connected and have a single root, that is, a single type that has no outgoing edges. This is the most general type, of which every other type is a sub-type. In Wikidata, the root type is *Q35120:entity*. Types that have no incoming edge are called the leaves or leaf-types. Figure 2.1 presents a small example taxonomy.

2.1.3 Relations

Entities allow KBs to record the existence of anything. These entities can then be organized using types. A final missing step in order to record knowledge is to record links and interactions between entities. This is the role of relations.

Definition 7 (Relation)**.** *Given a set of entities $\mathcal{E}$, a relation (or predicate) is a semantically meaningful subset of the Cartesian product $\mathcal{E} \times \mathcal{E}$. The subsets of $\mathcal{E}$ on which the relation is defined are called its domain and range.*

An intuitive way to understand this definition is to see relations as semantic links between types. For example, the types *person* and *city* might define the relation *placeOfBirth* as a subset of the Cartesian product *person* $\times$ *city*. It will contain couples of a person and their city of birth, e.g. $(Freddie\ Mercury, Stone\ Town) \in placeOfBirth$. This is just an intuition, however, because in general relations can be defined on any subset of $\mathcal{E} \times \mathcal{E}$, not only on types.

Just like entities, relations are referred to by unique identifiers and labels. For example, relation identifiers in Wikidata are formed by and integer preceded by the letter P: *P19* is the identifier of the *placeOfBirth* relation.

There exists a broader definition of relations where the Cartesian product involves more than two sets of entities. The number of sets of entities involved is called the *arity* of the relation. The previous definition concerns relations of arity 2: binary relations. In the thesis, only binary relations will be used.

Definition 8 (Triple)**.** *Given a set of entities $\mathcal{E}$, r a relation on $\mathcal{E}$ and $(e_1, e_2) \in \mathcal{E} \times \mathcal{E}$, $\langle e_1, r, e_2 \rangle$ is called a triple.*

Definition 9 (Fact)**.** *Given a set of entities $\mathcal{E}$ and a set of relations $\mathcal{R}$ on $\mathcal{E}$, the triple $\langle h, r, t \rangle \in \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ is a fact if $(h, t) \in r$, otherwise it is an unknown triple.*

The arguments of a relation are respectively called the *head* entity (noted h) and the *tail* entity (noted t). Subsequently, a triple is either a *fact* or an *unknown triple*. The next section will present the various assumptions under which unknown triples can

be handled. As stated earlier, types are entities as well. It is then possible to define a relation *hasType* such that given a type T, $e \in T$ can be noted $\langle e, hasType, T \rangle$.

Notation remark The notations presented here are the ones usually used by the machine learning community. They differ from the ones used by the semantic web community who calls triples *statements*, denoted *p(s,o)*. A statement is made of a predicate p, a subject s and an object o, respectively corresponding to a relation, a head entity and a tail entity.

Definition 10 (Inverse relation)**.** *Given a set of entities $\mathcal{E}$ and r a relation on $\mathcal{E}$, the inverse relation of r is a relation r^{-1} such that $\forall (x, y) \in \mathcal{E}^2, (x, y) \in r \iff (y, x) \in r^{-1}$.*

Definition 11 (Function)**.** *A function is a relation that has for each head at most one tail.*

Typical examples for functions are *birthPlace* and *hasLength*: people have at most one birthplace and every river has at most one length. Some relations are *nearly functions*, in the sense that very few heads have more than one associated tail. For example, most people have only one *nationality*, but some may have several. This idea is formalized by the notion of *functionality* [114]. The functionality of a relation r is the number of heads, divided by the number of facts involving that relation. It is always a value between 0 and 1, and it is 1 if r is a function. It is undefined for an empty relation.

$$\begin{aligned} fun\colon \mathcal{R} &\to [0, 1] \\ r &\mapsto \frac{|\{h : \exists t : \langle h, r, t \rangle \text{ is known}\}|}{|\{h, t : \langle h, r, t \rangle \text{ is known}\}|} \end{aligned}$$

When building KBs, it is usually possible to have to choose between a relation and its inverse relation, for example *isCitizenOf* or *hasCitizen* for the relation linking countries and their citizens. In general, KBs tend to choose the relation with the higher functionality, i.e. where the head has fewer tails. The intuition is that facts should be *about the head*.

2.1.4 Knowledge base

Using the previously defined entities, types and relations, a knowledge base can now formally be defined.

Definition 12 (Knowledge Base)**.** *Given, a set of entities $\mathcal{E}$ and a set of relations $\mathcal{R}$ on $\mathcal{E}$, a Knowledge Base (KB), in its simplest form, is a set of facts involving entities of $\mathcal{E}$ and relations of $\mathcal{R}$.*

The general KBs are the most well-known and notorious examples are Wikidata, DBpedia and YAGO [127, 74, 115]. A review of existing KBs is proposed in Section 2.2.

Definition 13 (Completeness)**.** *A knowledge base is complete if it contains all entities and facts of the real world in the domain of interest. The complete knowledge base is usually noted $\mathcal{K}^*$.*

Definition 14 (Correctness)**.** *A knowledge base $\mathcal{K}$ is correct if $\mathcal{K} \subseteq \mathcal{K}^*$, where $\mathcal{K}^*$ is the complete KB.*

In real life, large KBs tend to contain some erroneous triples. For example, the authors or YAGO4 [101] estimate that 95% of its triples are in $\mathcal{K}^*$, which is in practice approximated by Wikipedia. This means that YAGO4 still contains hundreds of thousands of wrong triples. For most other KBs, the degree of correctness is not even known.

An important remark is that KBs usually do not model negative information, but only *positive* facts. For example, Wikidata may say that Eurostar serves the cities of

Paris and London but it will not record that this train service does not serve the city of Stockholm. While incompleteness tells us that some facts may be missing, the lack of negative information prevents us from specifying which facts are missing because they are false. This poses considerable problems, because an unknown triple might as well be considered false or missing and it does not allow any conclusion about the real world [103].

Subsequently, there are two ways to consider missing information from KBs. First, the Closed World Assumption (CWA) considers the KB at hand as complete. This implies that any triple that is not in the KB is not in $\mathcal{K}^*$ either and is subsequently false. However, KBs are hardly ever complete. Therefore, KBs typically operate under the Open World Assumption (OWA), which says that if a triple is not in the KB, then it can either be true or false in the real world.

Definition 15 (Axiom)**.** *An axiom is a logical constraint that usually come with a KB.*

For example, it can be imposed that if e has the type t, and if t is a sub-type of t', then e must also have the type t': $\langle e, hasType, t\rangle \wedge \langle t, subTypeOf, t'\rangle \Rightarrow \langle e, hasType, t'\rangle$. Usual axioms are the following:

- **Domain (resp. range) constraints** force the head (resp. tail) entities of a relation to have a certain type. For example the domain of *bornIn* is included in the type *human* and its range is included in the type *place*.
- **Cardinality constraints** limit the number of tail entities per head for a certain relation. For example, entities of type *human* can only be involved in one fact with the relation *dateOfBirth*.
- **Symmetry, transitivity, and inverse constraints** force a relation to be symmetric, transitive, or the inverse of another relationship.
- **Subsumption constraints** force a relation to imply another one. For example *isMarriedTo* should always imply *isContemporaryTo*.
- **Disjointness constraints** force types not to have entities in common, For example: places and people.

Such axioms exist in packages of different complexity: the Resource Description Framework Schema (RDFS) is a system of basic axioms that are concerned mainly with types of entities. The axioms are so basic that they cannot result in contradictions. The Web Ontology Language (OWL) is a system of axioms that exists in several flavors, from the simple to the undecidable [116]. Such packages of axioms, together with the taxonomy, can be called *ontology* or *schema*.

2.1.5 The semantic web

The common exchange format for KBs is RDF, which stands for Resource Description Framework [134]. It specifies a syntax for writing down facts with binary relations. Most notably, it prescribes Uniform Resource Identifiers (URIs) as identifiers, which means that entities can be identified in a globally unique way. For example the URIs of Wikidata entities are the concatenation of the Wikidata URL with the entity identifier, for example `https://www.wikidata.org/entity/Q15869`.

To query such RDF KBs, the most appropriate query language is SPARQL [135]. It borrows its syntax from SQL, and allows the user to specify graph patterns, i.e. triples where some components are replaced by variables. It is common for public KBs to provide a public endpoint, which can be queried. For example, one can ask Wikidata[1] for the birth date of Freddie Mercury by writing the SPARQL query of Figure 2.2 where $P569$ is the identifier of the *birth date* relation and $Q15869$ is the identifier of the *Freddie Mercury* entity.

[1] `https://query.wikidata.org`

```
SELECT ?birthdate
WHERE
{
    wd:Q15869 wdt:P569 ?birthdate
}
```

Figure 2.2: SPARQL query asking the birth date of Freddie Mercury to Wikidata. $P569$ is the identifier of the *birth date* relation and $Q15869$ is the identifier of the *Freddie Mercury* entity.

	# entities	# relations	# facts	share of typed entities
Wikidata	101M	10,775	14.5B	94%
YAGO4	67M	116	343M	100%
DBpedia[3]	6M	3,253	9.5B	87%

Table 2.1: Descriptive figures of Wikidata, YAGO4 and DBpedia.

Many KBs are publicly available online and form what is known as the *Semantic Web*. Some of these KBs talk about the same entities, though with different identifiers. The Linked Open Data project[2] [12] establishes links between equivalent identifiers, thus weaving all public KBs together into one giant knowledge graph. The following section introduces a few of the existing notorious KBs.

2.2 Strengths and limits of existing knowledge bases

Obviously Wikidata is an example of a public knowledge base. It currently counts a hundred million entities and it is growing fast thanks to nearly twenty-four thousand active contributors. Despite being the best-known project, it was not the first. As explained in [116], the Cyc (1993) and the WordNet (1998) projects [85, 44] were among the first open-source ones to tackle the tedious challenge of centralizing knowledge, followed notably by KnowItAll and ConceptNet in 2004 [42, 79], YAGO and Freebase in 2007 [115, 14], BabelNet and Wikidata in 2012 [92, 127] and DBpedia in 2015 [74]. Among these, Wordnet, Freebase and DBpedia are some of the most widespread in machine learning literature, though Freebase was discontinued in 2016 in favor of Wikidata and Wordnet was not updated since 2011. Table 2.1 reports the sizes of Wikidata, YAGO4 and DBpedia.

Currently, the largest public KB is Wikidata. Its large set of contributors makes it a very abundant collection of information. Though Wikidata does not have a strong definition of the concept of types, they can be defined empirically as the entities of the KB that are involved as tails of facts with the relations *P31: instance of* or *P279: subclass of*. The latter one also defines the subsumption binary relation on types necessary to build a taxonomy. As of 2020, only 6% of the Wikidata entities were not typed[4] and according to [101], Wikidata contains approximately 2.4M types. However, there is a downside to the obvious richness of Wikidata: it is sparse. For example, 80% of the types have less than ten entities in the KB. Another issue can come from the structure of the taxonomy itself: it is not a tree and because of the lack of constraints, for a single type, the numerous paths up to a more general type can diverge in meaning. For example, the *happiness* entity has the type *emotion*. There are several possible paths from *emotion* to the root entity in the Wikidata taxonomy. The shortest one might be the most intuitive: *entity/qualia/emotion/basic emotion*. Another possible path counts fifteen types: *entity/collective entity/set/group/series/activity/behavior/human behavior/human activity/intentional human activity/use/function/mental process/emo-*

[2] https://lod-cloud.net
[3] https://www.dbpedia.org/blog/yeah-we-did-it-again-new-2016-04-dbpedia-release/
[4] https://www.wikidata.org/wiki/Wikidata:Statistics

tion/basic emotion. In a 2020 archive, we counted ten different such paths. This can be the source of errors and incoherence as it can make machine learning methods noisy by concealing patterns.

Contrasting with this very complex and unconstrained taxonomy, schema.org[5] proposes a much simpler and constrained taxonomy. This is an unavoidable contribution to the semantic web and currently, the taxonomy counts 797 types and 1,457 properties describing the types. Some extensions of the taxonomy are proposed on specific topics in order to properly match specialized KBs. A major extension is the "*health and life science*" one which accounts for 80 types and 162 properties.

Attempting to combine the best of both projects, YAGO4 [101] was introduced in 2020 to structure the profusion of entities in Wikidata with the rigorous design of the schema.org taxonomy. The design of the KB relies precisely on entity typing with a hybrid taxonomy: the top-level comes from schema.org, whereas the leaf types come from Wikidata. The junction mapping was done manually. Eventually, for each entity in Wikidata, if the path to *entity* in the original taxonomy starts with a type that has a Wikipedia page then the type is kept and is mapped to the schema.org type that first appears on the path up to *entity*. Eventually, types with less than 10 entities were discarded and so were the corresponding entities. The resulting taxonomy counts ten thousand types and the largest version of the KB contains 67 million entities and 343 million facts.

Those are publicly available KBs but there are also a lot of private ones (and certainly even more). Big tech companies of course maintain their own knowledge graphs. Google and Microsoft's search engines enrich results with facts extracted from their KBs and Apple and Amazon use their own ones to help their personal assistants answer queries.

2.3 From knowledge bases to knowledge graphs

The previous section explains how knowledge can be represented and recorded in the form of knowledge bases. These are often studied by looking at their natural graph structure. Let us first define rigorously what a graph is.

2.3.1 Graph definitions

A graph is a mathematical object that represents the links between entities. In its simplest form, a graph is a list of n nodes or vertices $V = \{v_1, v_2, \cdots v_n\}$ and p *edges* between them $E = \{e_1, e_2, \cdots e_p\} \subset (V \times V)^p$. It is usually denoted $G = (V, E)$. A more general version allows edges to carry weight.

Definition 16 (Adjacency matrix)**.** *Given a graph $G = (V, E)$ and $\mathbb{K}$ a field, G is fully defined by the adjacency matrix $A \in \mathbb{K}^{n \times n}$:*

$$A_{ij} = \begin{cases} w_{i,j} \in \mathbb{K} \text{ if } (i,j) \in E, \\ 0 \text{ otherwise.} \end{cases}$$

with $w_{i,j} \in \mathbb{K}$ the weight of edge (i, j).

The adjacency matrix of an unweighted graph is a binary matrix belonging to $\{0, 1\}^{n \times n}$.

Depending on the form of the adjacency matrix, a graph can present some topological features. There are mainly three types of graphs:

- undirected graphs have a symmetric adjacency matrix ($A = A^T$).
- directed graphs have a non-symmetric adjacency matrix ($A \neq A^T$).

[5] https://schema.org

- bipartite graphs have an adjacency matrix that can be written in the form of a block matrix up to a reordering of the nodes, as in Equation 2.1 where $k \in \{1, \dots, n-1\}$ and $B \in \mathbb{K}^{k,n-k}$.

$$A = \begin{pmatrix} 0 & B \\ B^T & 0 \end{pmatrix} \tag{2.1}$$

  Intuitively, nodes in a bipartite graph can be divided in two groups and edges can only link nodes from one group to nodes of the other.

Definition 17 (Degree matrix)**.** *Let $G = (V, E)$ be a weighted undirected graph and A its adjacency matrix. Then $\boldsymbol{d} = A \cdot \mathbb{1}$ are the node degrees and $D = \mathrm{diag}(\boldsymbol{d})$ is called the degree matrix of G.*

A graph is said to be connected if there exists a path of edges between any pair of nodes. In directed graphs, strong connection involves directed edges whereas weak connection removes the direction of edges. Connection (resp. weak connection) defines an equivalence relation on the nodes of a graph (resp. directed graph). The equivalence class of these are called the connected components (resp. weakly-connected components) of the graph.

2.3.2 Graph structure of knowledge bases

As presented in Section 2.1.4, a KB is a collection of facts, i.e. known triples linking a head entity to a tail entity by a relation. Facts can naturally be seen as edges in a graph where the nodes are the entities. This graph structure of a KB is referred to as a Knowledge Graph (KG).

Definition 18 (Knowledge graph)**.** *A Knowledge Graph (KG) is a triple $(\mathcal{E}, \mathcal{R}, \mathcal{F})$ such that $\mathcal{E}$ is a set of entities, $\mathcal{R}$ a set of relations on $\mathcal{E}$ and $\mathcal{F}$ a set of facts of $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$. It is usually noted $\mathcal{K}$.*

Using the usual graph terminology, a KG is a directed graph with labels on the edges. The nodes are the entities and the labels on the directed edges are the relations.

A graph structure is naturally formalized as an adjacency matrix. Likewise, the KG structure can easily be formally written in the form of 3-dimensional adjacency tensor.

Definition 19 (Adjacency tensor)**.** *Given a KG $(\mathcal{E}, \mathcal{R}, \mathcal{F})$, its adjacency tensor is defined as $\mathcal{T} \in \{0, 1\}^{n \times n \times m}$ such that*

$$\mathcal{T}_{i,j,k} = \begin{cases} 1 \text{ if } (e_i, r_k, e_j) \in \mathcal{F} \\ 0 \text{ otherwise.} \end{cases}$$

where $n = |\mathcal{E}|$, $m = |\mathcal{R}|$, $\mathcal{E} = \{e_1, \dots, e_n\}$ and $\mathcal{R} = \{r_1, \dots r_m\}$.

Intuitively, the adjacency tensor of a KG is the concatenation along the third dimension of the adjacency matrices of the graphs containing the edges of only one relation. Figure 2.3 shows a 3D visualization of an adjacency tensor.

Terminology remark The definitions of knowledge bases and knowledge graphs are very close. The first one historically comes from the field of semantic reasoning, while the second one is more prevalent in the field of machine learning. The main difference is that *knowledge graph* refers to the graph structure of the fact collection of a *knowledge base*. Commonly, knowledge base implicitly refers to the fact collection along with the ontology. In the rest of this thesis, the term *knowledge graph* will be mostly used.

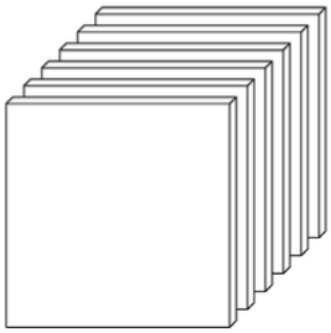


Figure 2.3: Frontal concatenation of adjacency matrices. Each slice of the tensor can be seen as the adjacency matrix corresponding to a specific relation.

Chapter 3

Knowledge graph completion

The previous chapter introduced the concept of knowledge base and the graph structure of the fact collection, the knowledge graph.

As explained in the introduction, researchers have tried to automatize several tasks related to KB construction and maintenance. Depending on the stage of development of the KB, the task might be very different. A very early-stage task could be to structure the knowledge contained in a text corpus (e.g. Wikipedia pages). This might require the use of natural language processing techniques for example. If the KB is at a more advanced stage of development, it might already have reached a critical size and contain enough information for an agent to partially complete it without requiring external sources of knowledge. For example, a KB containing people's place of birth could be enriched with facts on people's nationality.

Completing a knowledge base essentially comes down to suggesting new facts. There has been a variety of proposed solutions to solve this task automatically. The two main fields are semantic reasoning and machine learning. Coherently with the final remark of the preceding chapter, semantic reasoning usually tackles knowledge *base* completion while machine learning tackles knowledge *graph* completion. The task is however the same in both cases. In the current chapter, a brief introduction to machine learning and representation learning is proposed followed by a state of the art of machine learning completion methods of KGs. Eventually, the semantic reasoning methods are briefly introduced along with methods combining the two approaches.

3.1 Introduction to graph representation learning

Machine Learning (ML) is a field of computer science that develops algorithms to automatically learn which aspects of the data are relevant to solve a specific task. It is considered as a part of artificial intelligence and originally relied on statistics to mine patterns in the data.

A demonstrative example of a task that can be solve with a ML algorithm is spam detection. Internet users often receive a large number of emails that they wish they did not receive like advertisement or phishing attempts. A way to filter the unwanted emails could be to manually maintain a list of specific keywords and to dismiss incoming emails that contain any of these. A more comfortable solution is let an algorithm maintain the list by detecting which keywords are characteristic of spam emails. A vanilla version of this algorithm would *learn* the set of banned keywords on a set of annotated emails but a more complex version should update the rules continuously as the user flags new emails.

KGs can be considered as annotated data. As stated earlier, they model positive information and thus facts come with a *True* label. A natural task would thus be to build and train a model to classify triples as *True* or *False*. The problem is that, as stated in Chapter 2, KBs do not model negative information and training a ML algorithm with only one label does not make much sense. For the algorithm to learn the difference between the labels, it has to be shown true and false triples. This problem of false triple generation, called Negative Sampling, will be presented in Section 3.2.2. Ignoring it for now, let us focus on supervised ML.

3.1.1 Supervised machine learning

A formal definition of supervised ML is the following. Let $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} \in (\mathbb{R}^k)^n$ be a dataset of n samples with k features. Each sample $\boldsymbol{x}_i$ comes with a label $y_i \in \mathbb{R}$. Let $\boldsymbol{y} \in \mathbb{R}^n$. This $(X, \boldsymbol{y})$ pair naturally defines a supervised task: defining a function $\phi : \mathbb{R}^n \to \mathbb{R}$ fitting the annotated samples, i.e. $\phi(\boldsymbol{x}_i) = y_i$ for as many $i \in \{1, \dots, n\}$ as possible. This is done hoping that ϕ generalizes properly to label correctly unseen data samples. Depending on the possible values for the labels, the task has a different name:

- if the set of possible labels is finished, then the task is called *classification*.
- if the set of possible values is a range in $\mathbb{R}$, then the task is called *regression*.

Solving this task relies on the assumption that there exists a true labelling function $\phi_0 : \mathbb{R}^n \to \mathbb{R}$ such that $\forall i \in \{1, \dots, n\}$, $\phi_0(\boldsymbol{x}_i) = y_i$. This function is called a ground truth oracle. Supposedly, the annotations of the dataset were generated by it and given any new sample, it can predict its correct label.

There are possibly many reasons for which it is impossible to call the oracle ϕ_0 every time an annotation is needed. For example, in image processing, asking the oracle for an annotation often comes down to asking a pool of human annotators to describe the content of an image, which is very costly. Physics is also full of examples of expensive calls to the oracle that often amount to solving complex equations at the cost of very heavy computations [6].

A ML algorithm can then be defined as a parametric function $\phi : \mathbb{R}^n \to \mathbb{R}$ that approximates ϕ_0 and that is *hopefully* much easier to be called on unseen samples. *Training* this algorithm comes down to adjusting the parameters of ϕ so that the known annotations are fitted, i.e. $\phi(\boldsymbol{x}_i) = y_i$ for as many $i \in \{1, \dots, n\}$ as possible. Though the form of the function is typically defined by a human, the optimization of its parameters is done by the machine. ϕ is usually called a ML *model* because the engineering of the function should reflect a deep understanding of the structure of the data and require modelling skills. Two fundamental characteristics of ML models are the following:

- the **expressive power** can intuitively be defined as the amount of information the model can store and it is directly linked to its number of parameters.
- the **generalization capacity** of a model refers to its ability to adapt to unseen data samples and more precisely to data points that diverge from the distribution of the training samples.

There is a natural trade-off between the two. Intuitively, a ML model needs to be given enough expressive power to record possibly complex patterns from the data. However, too much expressive power can lead to a model that is difficult to train (because of too much parameters) and that generalizes poorly because the numerous parameters fit *too well* the training samples. The latter phenomenon is called overfitting and is controlled by regularization. Appendix A presents three common regularization techniques and lists references from [52] for extensive analysis. In practice, ML models almost never fit perfectly the training data.

In a simple binary classification task of data in $\mathbb{R}^2$, the *learned* pattern is simply a border separating the plane in regions with label 0 and others with label 1. Figure 3.1

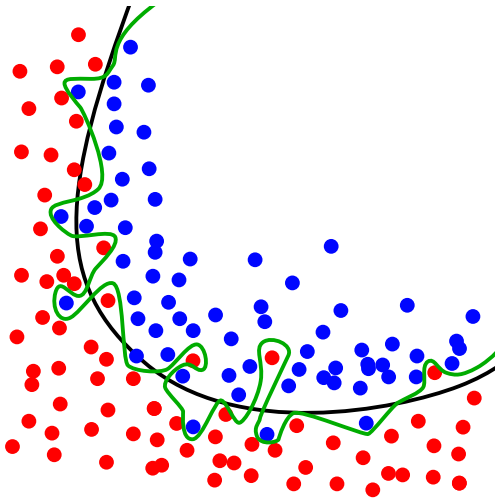


Figure 3.1: Visualization of overfitting in the case of 2D binary classification (source of the image: Wikipedia[1]). Samples are colored in blue or red depending on their label. The green border corresponds to an overfitting model. The black border might generalize better to new data points.

illustrates the overfitting phenomenon with two possible borders.

A widespread naming convention in ML calls *parameters* the variables of a model that are adjusted during training to fit the data. The other variables that are used to define the form of the function ϕ but that remain unchanged afterwards are called *hyper-parameters*.

An important problem in ML is model selection: given several models (possibly the same one but with different hyper-parameters), choosing the best performing one can be tricky. Commonly, the best model is defined as the one that performs the best on previously unseen data. This can be estimated by measuring performance metrics on a test set distinct from the training set. Because of overfitting, the best model might not be the one performing best on training data. When some hyper-parameters must be chosen, it might be tempting to select the ones resulting in the best test performances. This can however lead to overfitting as well. A common technique to properly select the hyper-parameters is k-fold cross validation [56]: it splits the training set in k groups and independently trains the model k times. Each time, one of the groups is used as validation set to evaluate the performance of the model while the $k-1$ groups are used as training data. Eventually, the performance of the model is set to the average validation performance of the k independent runs. This value then can be used to select the hyper-parameters. More details can be found in Chapter 1 of [9].

Let us conclude this section with a simple remark which will be useful later in the chapter. When solving a supervised ML problem, the available training data should contain at least two different labels. Otherwise, a simplistic model always outputting the only training label could perfectly fit the data while having a very poor generalization capacity. An intuitive way of looking at supervised ML models is to think that those models are trained to make differences, to learn distinctive patterns in the features describing the samples.

3.1.2 Representation learning

ML models need to represent real-world objects in the form of vectors to process them. Often, the representation of objects is natural from the problem definition. In the Iris

[1] https://en.wikipedia.org/wiki/Overfitting

dataset for example [46], flowers are represented by four of their measures. However, sometimes there is no natural representation available of the data or the only one is unsatisfactory, because it is in too high dimension for example (cf. one-hot encoding in Definition 20). An entire field of ML focuses on learning representations of objects: it is called *Representation Learning*. This is usually only a means to an end but it is a necessary step to solve real-world problems.

The simplest way to represent a set of objects as vectors is by a one-hot encoding:

Definition 20 (One-hot encoding)**.** *Given an ordered set of objects $S = \{o_1, \dots, o_n\}$, the one-hot encoding of an object o_i is the vector $\boldsymbol{h}(o_i) \in \{0,1\}^n$ that contains only zeros, and a single 1 at position i.*

This is a very poor representation as it only contains the information that samples are distinct from one another. It does not carry descriptive information. It should be accompanied by additional data, such as an adjacency matrix if the objects are nodes of a graph.

Definition 21 (Embedding)**.** *A d-dimensional embedding for a group of objects (e.g. words, entities) is an injective function that maps each object to a vector in $\mathbb{K}^d$, so that the intrinsic relations between the objects are maintained. $\mathbb{K}$ is a field and usually $\mathbb{K} = \mathbb{R}$.*

This definition is intentionally vague on what the intrinsic relations are. The main interest of representing objects as vectors comes with the metric structure of vector spaces allowing to measure distances between objects. The difficulty lies in the interpretation of that distance and this is linked to the vague notion of *intrinsic relations* used in Definition 21. For example, *closeness* between words can be related to their semantic proximity or to their grammatical use or the two altogether. For entities and relations of KGs, they can be considered close if they often appear in facts together or if the entities are semantically close in the real world. The design of the embedding should result in a metric that can be interpreted in term of this closeness.

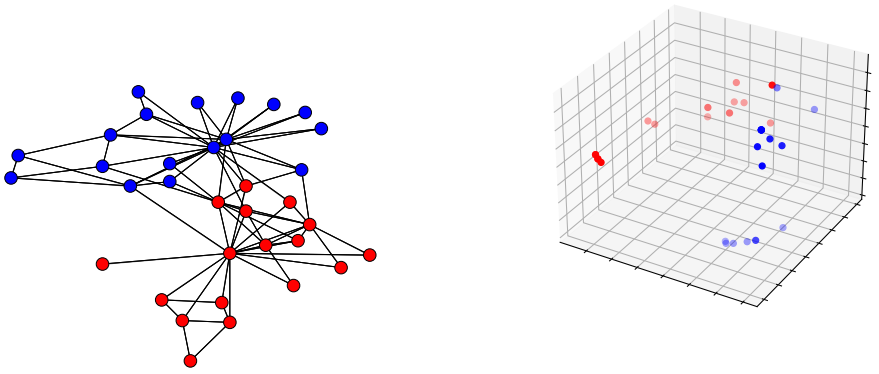
Obviously, a one-hot encoding could be considered as a n-dimensional embedding. A goal of embedding techniques however usually is to reduce the embedding dimension to the minimum to ease the processing of the resulting vectors in downstream tasks. Scalability and computational cost are obvious concerns, but more refined problems can appear in high dimensions. A common one is the *curse of dimensionality*. The expression, which was coined by Bellman in 1957 usually refers to the fact that the volume of a space grows exponentially with its dimension. This leads to data becoming sparse and counter-intuitive phenomena arising.

As previously defined, the function used to compute the vector representations is called the embedding or the embedding model. By extension, the resulting vectors are usually called the embeddings of the object or the embedding of the data. Finally, the field dealing with the study of embedding models is called representation learning.

Learning representations can either be done upstream of the ML task at hand (if no natural representation of the data is naturally available for example) or it can be done simultaneously. A classical way to embed a set of objects is to train a parametric ML model on a task linked to the *closeness* one wants to capture in the embedding. Once trained, an embedding of the objects can possibly arise in the parameters if they can be mapped to the objects.

For example, the OpenFlight dataset[2] records international airports and daily flights linking them. This is a weighted graph, the nodes of which are airports. An embedding of the airports representing highly linked airports by close vectors could be computed by training a ML model to predict the number of daily connections between airports.

[2] https://netset.telecom-paris.fr/pages/openflights.html



(a) Karate club graph. (b) Embedding of the same graph in $\mathbb{R}^3$

Figure 3.2: Visualizations of the embedding of the Karate club graph [145] in $\mathbb{R}^3$ using spectral embedding.

3.1.3 Graph embedding with spectral theory

When it comes to graphs, there exists a variety of representation learning tasks in the literature. The objects to embed can be either the nodes of a graph or a set of graphs altogether. The latter case is common in biology: some molecules can be represented as graphs and embedding those can make molecules comparable by a simple distance measure. The common and best-documented task however is the problem of node embedding for which the distance between vectors can be interpreted as how the original nodes are likely to be linked by an edge. The current section presents an example of such a technique, relying on spectral theory.

Let $G = (V, E)$ be a graph, n the number of nodes, A the adjacency matrix and D the degree matrix. A d-dimensional embedding for this graph consists in vectors $(\boldsymbol{x}_i)_{i\in\{1,\dots,n\}} \in (\mathbb{R}^d)^n$ representing the nodes. Let $X \in \mathbb{R}^{n\times d}$ be a matrix in which the i-th row is $\boldsymbol{x}_i$. Encoding the structure of the graph in X can be done for example by forcing that two *close* nodes i and j in the graph be represented by *close* vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in $\mathbb{R}^d$ (cf. Figure 3.2).

An intuitive way to achieve this is to design a model that can estimate the likelihood of an edge to exist between any pair of nodes using the distance between their embeddings. Let us write this model as the parametric function f_X:

$$\begin{array}{rccc} f_X & : & V\times V & \to & \mathbb{R} \\ & & (i,j) & \mapsto & ||\boldsymbol{x}_i - \boldsymbol{x}_j|| \end{array}$$

As explained previously, this parametric function can be interpreted as an approximation of the oracle that returns larger values for existing edges and smaller ones otherwise. The parameters X need to be adjusted by solving the following minimization problem:

$$\begin{aligned} &\min_{X\in\mathbb{R}^{n\times d}: X^T\mathbb{1}=0, X^TX=I_k} \sum_{(i,j)\in E} A_{i,j} \times f(i,j)^2 \\ &= \min_{X\in\mathbb{R}^{n\times d}: X^T\mathbb{1}=0, X^TX=I_k} \sum_{(i,j)\in E} A_{i,j} \times ||\boldsymbol{x}_i - \boldsymbol{x}_j||^2 \end{aligned} \tag{3.1}$$

There are two constraints on $X \in \mathbb{R}^{n\times d}$: $X^T\mathbb{1}$ forces the embedding to be centered on 0 and $X^TX = I_k$ forces the embeddings to have uncorrelated coordinates of positive variance. The latter constraint adds repulsive forces between the vectors and forces them to occupy the whole space, thus avoiding the trivial solution $X = 0$.

The problem in Equation 3.1 can be solved using spectral analysis of the matrix A. Let us briefly state the Spectral Theorem, which is necessary to express the solutions.

Definition 22 (Eigenvalue and eigenvector)**.** *Given a matrix $A \in \mathbb{K}^{n\times n}$, if $(\lambda, \boldsymbol{x}) \in \mathbb{K} \times \mathbb{K}^n$ is a solution of $A\boldsymbol{x} = \lambda\boldsymbol{x}$ then $\boldsymbol{x}$ is an eigenvector of A associated to the eigenvalue λ.*

Definition 23 (Graph Laplacian)**.** *Let $G = (V, E)$ be a weighted undirected graph, A its adjacency matrix and D its degree matrix. The Laplacian of G is defined as $L = D - A$ and its normalized version is defined as $\mathcal{L} = I - D^{-1/2}AD^{-1/2}$. L and $\mathcal{L}$ are real symmetric.*

Theorem 1 (Spectral theorem)**.** *Let $M \in \mathbb{R}^{n\times n}$ be a real symmetric matrix then there exists $P \in \mathbb{R}^{n\times n}$ and $\Lambda \in \mathbb{R}^{n\times n}$ such that $M = P \cdot \Lambda \cdot P^T$ and*

- *The columns of P are the eigenvectors of M and $P \cdot P^T = P^T \cdot P = I_n$.*
- *Λ is diagonal and its diagonal elements $(\lambda_1, \lambda_2, \ldots, \lambda_n)$ are the eigenvalues of M.*

Let us call P and $(\lambda_1, \ldots, \lambda_n)$ the spectral decomposition of M.

A demonstration of this theorem can be found in [75].

Let L be the Laplacian of G, which is symmetric. Let $\lambda_1 \leq \cdots \leq \lambda_n$ be the eigenvalues of L given by the Spectral theorem. The columns of the matrix X solving the problem in Equation 3.1 are the eigenvectors of the Laplacian matrix L associated with the eigenvalues $\lambda_2, \ldots, \lambda_{d+1}$. A demonstration of the result can be found in [15].

This gives a graph embedding method called Spectral embedding. A comprehensive presentation of the spectral analysis of graphs can be found in [28].

3.1.4 Extending spectral embedding to knowledge graphs

Though KBs are naturally represented as graphs, the spectral analysis cannot be used as is for KG embedding. Indeed, as explained in Section 2.3.2, a KG is represented by a 3-dimensional tensor and not as a matrix. The spectral embedding method relies on spectral analysis and eigenvalue decomposition of square matrices, which are particular cases of the study of Singular Value Decomposition (SVD) of non-square matrices. To apply to KGs, these have to be extended to higher-order tensors. According to Kolda et Bader [68], there are mainly two generalizations of the SVD: the CANDECOMP/-PARAFAC (CP) decomposition, introduced by Hitchcock in 1927 [57] and the Tucker decomposition [123]. This second method is often also presented as a generalization of the usual principal component analysis [100]. Explicit details of these techniques are beyond the scope of the current chapter but they will further be introduced indirectly with the presentation of DistMult and ComplEx algorithms [141, 121].

As explained by Kolda et Bader [68], the CP and Tucker decompositions pose some problems compared to the extensively studied SVD of matrices. For example, the dimensions of the decompositions of a higher-order tensor cannot be ranked according to reconstruction quality as it was done in Section 3.1.3 with a matrix. There are examples where the best rank-one approximation of a 3-dimensional tensor is not a factor in the best rank-two approximation. Those factors then have to be computed all-together and cannot be computed sequentially, which poses computation issues.

There are however several KG embedding techniques that were based on CP or Tucker decompositions. For example DistMult and ComplEx [141, 121] implement constrained versions of the CP decomposition of the adjacency tensor whereas TuckER [4] adapts the Tucker decomposition to the same task. Those methods will be detailed in the following section. An extensive comparison of CP decomposition for automatic KG completion can be found in [72].

3.2 Knowledge graph representation learning

KG representation learning has been a trending topic in recent years. Most of the available literature focuses on embedding entities and relations. To the best of our knowledge, no published work on embedding entire KGs as single objects has drawn significant attention. This section, introduces the main formalism used to describe KG embedding models relying on a scoring function. A variety of existing models are then reviewed.

3.2.1 Estimating triples likelihood

Despite the large number of publications focusing specifically on KG embedding, it is a mean to solve other tasks, and mainly the automatic completion of KGs.

Automatic completion of KGs is can be formalized as a supervised binary classification task: categorizing triples as true or false. The ground truth oracle here would be an omniscient intelligence with encyclopedic knowledge about the world, or at least the topic of the KG at hand, making annotations obviously difficult to get.

As stated earlier, most KGs do not model negative information, which means that the available training data only contains samples from one of the two labels. This is problematic as explained in the final remark of Section 3.1.1. Methods to get a hold of false triples will be detailed in the Section 3.2.2. Until then, let us suppose that true facts and false triples are available.

Approximating this oracle is often tackled by the close task of estimating the likelihood of a triple to be true. Models are then defined as a parametric scoring function that estimates likelihood.

Definition 24 (Scoring function)**.** *Given a KG $\mathcal{K} = (\mathcal{E}, \mathcal{R}, \mathcal{F})$, a scoring function is a parametric mapping $f : \mathcal{E} \times \mathcal{R} \times \mathcal{E} \to \mathbb{R}$ that maps triples $\langle h, r, t \rangle$ to real scores $f(h, r, t)$. The score should be high for facts and low for false triples.*

A KG embedding model is entirely defined by its scoring function f. Some parameters of the function are usually vectors representing each entity and each relation, which are by extension used as embeddings. The specificity of each model lies in how the interaction between entities and relations is algebraically formalized in order to compute the score of triples. The rationale behind this method is that a model capable of correctly predicting the truth value of a triple should have encoded the necessary information in the representation it has of the involved entities and relation.

As an example, let us now look at one of the simplest embedding model. TransE, proposed in 2013 by Bordes et al. [17], simply represents entities and relations as d-dimensional vectors and models relations as translation between entities. This implies that the associated scoring function has $d \times (n + m)$ parameters in the form of two matrices $E \in \mathbb{R}^{n \times d}$ and $R \in \mathbb{R}^{m \times d}$ where n (resp. m) is the number of entities (resp. relations). Given an entity $e \in \mathcal{E}$ and a relation $r \in \mathcal{R}$, the corresponding rows of E and R, noted E_e and R_r, are the embeddings of e and r. The scoring function is eventually simply defined as:

$$\begin{array}{rccc} f_{E,R} & : & \mathcal{E} \times \mathcal{R} \times \mathcal{E} & \to & \mathbb{R} \\ & & \langle h, r, t \rangle & \mapsto & ||E_h + R_r - E_t||_2 \end{array}$$

KG embedding models are usually grouped together according to the form of their scoring functions: the main ones are the translational, the bilinear and the deep-learning models. Models inside families distinguish themselves by imposing various constraints on the representations. A more precise presentation of various families of models is proposed in Section 3.3.

3.2.2 Negative Sampling

As explained, negative triples are required to train embedding models on the binary classification task. However, most KGs only record positive triples (facts), an example used in Section 2.1 was that a KG might record that the Eurostar serves the cities of Paris and London. It cannot however exhaustively list all the places that are not served by it. This is related to the open and closed world assumptions: under the CWA, a city which is not listed as served by Eurostar is assumed not to be. Under the OWA, however, it is not possible to conclude. Let us introduce several existing methods to generate negative samples.

Uniform Negative Sampling

First, it is obvious that the OWA makes it impossible to train a supervised binary classifier of triples. Indeed, given a KG $\mathcal{K} = (\mathcal{E} \times \mathcal{R} \times \mathcal{F})$, the OWA divides the set of all possible triples $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$ between the triples that are known to be true $\mathcal{T}$ and all the others for which no decision can be made.

The CWA however, makes it easy to solve this problem: if false triples are required, any triple which is not in $\mathcal{T}$ can work. This led to a simple and yet effective way of generating false triples from known facts using randomness, a process called Negative Sampling (NS). Proposed by Bordes et al. [17], the uniform NS technique generates triples from facts by replacing either the head or the tail by an new randomly chosen entity. If the resulting triple is unknown, it is then considered false under the CWA. The process is detailed in Algorithm 1. The new triple is called a negative sample.

Algorithm 1: Uniform Negative Sampling.

Input: $\langle h, r, t \rangle$, a fact
Output: $\langle h', r, t' \rangle$, a false triple
Data: $\mathcal{T}$, the facts in the KG

$(h', t') \leftarrow (h, t)$
while $\langle h', r, t' \rangle \in \mathcal{T}$ **do**
  $u \leftarrow$ uniform random variable on $[0, 1]$
  if $u < \frac{1}{2}$ **then**
    $h' \leftarrow$ random entity
  else
    $t' \leftarrow$ random entity
return $\langle h', r, t' \rangle$

An underlying paradox

The simple definition of NS however hides a paradox. Under the CWA, a binary classifier could be trained by assuming that any unknown triple is false but would be used in the end to complete the KG with unknown triples likely to be true and yet previously supposed false. An ideal NS technique would generate false triples that are both very likely to be true and yet for sure false. For example, replacing *Amiens* with *Paris* in the fact $\langle EmmanuelMacron, wasBornIn, Amiens \rangle$ leads to a very likely triple: Emmanuel Macron could have been born in Paris. However, as it is a known fact that he was born in Amiens: the triple $\langle EmmanuelMacron, wasBornIn, Paris \rangle$ is for sure false. This is linked to the Partial Completeness Assumption (PCA), or Local Closed World Assumption (LCWA) introduced in [49]. It states that if a KB contains the facts $\langle h, r, t_1 \rangle$, ..., $\langle h, r, t_n \rangle$ then any triple $\langle h, r, t' \rangle$ with $t' \notin \{t_1, \ldots, t_n\}$ can be assumed false. The intuition behind this assumption is that if some contributor of the KB made the effort to document the entities linked to h by r, then it did it exhaustively. [48] showed that this is generally true for relations with fewer objects, such as *wasBornIn*. It has been

observed that generally, relations have a lower number of tail per head than head per tail. This observation argues in favor of an informed choice when selecting whether the head or the tail will be replaced in the NS procedure.

Bernoulli Negative Sampling

Introduced in 2014 by Wang et al. [131], this technique (noted BerNS) relies on relation-specific Bernoulli distributions to choose which entity between the head and the tail of a fact should be replaced to maximize the probability of the resulting triple to be false. Formally, a Bernoulli parameter p^r is computed for each relation r as follows:

$$p^r = \frac{\rho^r_{t,h}}{\rho^r_{t,h} + \rho^r_{h,t}},$$

where $\rho^r_{t,h}$ (resp. $\rho^r_{h,t}$) is the average number of tail entity per head entity (resp. head entity per tail entity) among all known facts involving r. This parameter p^r is the probability to replace the head entity of the fact.

Algorithm 2: Bernoulli Negative Sampling (BerNS).

Input: $\langle h, r, t\rangle$, a fact
Input: p^r, Bernoulli parameter for relation r
Output: $\langle h', r, t'\rangle$, a false triple
Data: $\mathcal{T}$, the facts in the KG

$(h', t') \leftarrow (h, t)$
while $\langle h', r, t'\rangle \in \mathcal{T}$ **do**
  $u \leftarrow$ uniform random variable on $[0, 1]$
  if $u < p^r$ **then**
    $h' \leftarrow$ random entity
  else
    $t' \leftarrow$ random entity
return $\langle h', r, t'\rangle$

As an example, consider *author of*, which is a one-to-many relation (one author and many potential books). In that case, the head entity (an author) should be more likely replaced than the tail entity (a book), yielding a false triple with greater probability.

Positional Negative Sampling

Introduced by Socher et al. in 2013, this NS technique simply constrains the random choice of new entities so that it respects domain and range constraints of relations. It is done without looking at the ontology. Given a fact $\langle h, r, t\rangle$, a replacement entity for the head (resp. tail) for example would be chosen at random among all the entities of the KG that are known to have been involved as a head (resp. tail) of a fact featuring the relation r.

Adversarial Negative Sampling

A more recent approach looks at NS as an adversarial problem. In 2017, Cai et al. proposed in [26] to train two embedding models concurrently. The main one acts as a discriminator: it should become able to discriminate between true and false triples that are presented to him. That is the usual binary classification task. Another model, playing the role of generator, is used to compute a likelihood distribution on a set of candidate triples generated using randomness. The sampling is eventually done using this distribution, making triples with high scores more likely to be generated. In 2019, Zhang et al. proposed in [148] to refine this adversarial approach by keeping track with a smart caching system of negative samples with high scores, which they claim are rare.

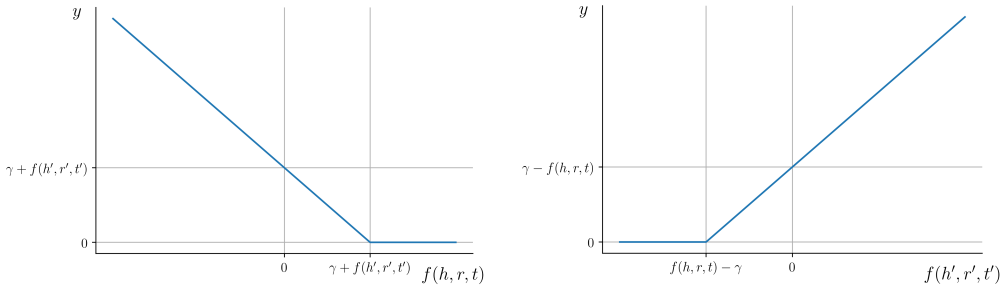


Figure 3.3: Margin loss as a function of the score of a true fact (a) and of the score of its negative version (b). In both cases $y = \max\{0, \gamma - f(h, r, t) + f(h', r', t')\}$.

To conclude on NS, it is interesting to remark that many proposed implementations accompanying KG embedding literature do not check whether the generated triples belong to the KG or not. This is supported by the sparseness of the KG, lowering the probability of finding an existing triple when the new entity is selected at random (even lower when using BerNS) as opposed to the computation cost of checking whether the new triple is already known.

3.2.3 Model training

Training an embedding model comes down to finding proper values for its parameters (the embeddings) so that the scoring function gives high scores to facts and low scores to false triples. This is usually done by minimizing an overall measure of the mistakes made on triples known to be true and their corresponding negative samples, supposed to be false. Given a training fact denoted $\langle h, r, t\rangle$, the corresponding false triple $\langle h', r', t'\rangle$ is generated by NS. Then for each resulting pair, a loss measuring the gap between the corresponding scores is computed, $\ell(f(h, r, t), f(h', r', t'))$. This loss ℓ should be high for close scores. These measures are called loss functions and the two historic ones are the margin loss (3.2) and the logistic loss (3.3).

Once again, it is necessary to involve true and false triples in the training procedure. If it were to involve only true facts, a trivial model classifying all triples as true could minimize any measure of performance. Intuitively, if we want the model to learn the difference between the true and false, it should be shown both. In the following, $\mathcal{F}$ is a set of training facts and $\mathcal{F}'$ is a negatively sampled version of it such that to each fact $\langle h, r, t\rangle \in \mathcal{F}$ corresponds its corrupted version $\langle h', r', t'\rangle \in \mathcal{F}'$.

Margin loss

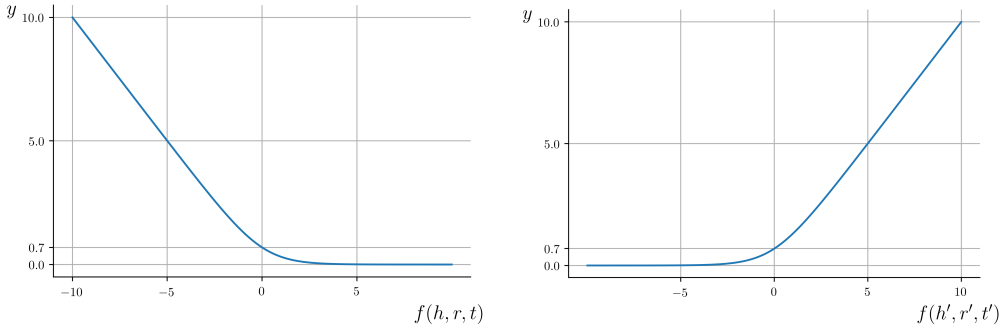
This loss separates the scores of true and false triples by a *margin* γ, which is a hyper-parameter. Once the difference between the scores of a fact and its corrupted version is larger than the margin, the loss is 0. The graphical shape of the loss is represented in Figure 3.3.

$$\mathcal{L} = \sum_{(h,r,t)\in\mathcal{F}} \max\{0, \gamma - f(h, r, t) + f(h', r', t')\} \tag{3.2}$$

Logistic loss

This loss associates low scores to false triples and high scores to true facts. Once the score of a true (resp. false) triple is above 3 (resp. below -3), the loss is approximately 0. The shape of the loss is represented in Figure 3.4.

$$\mathcal{L} = \sum_{\langle h,r,t\rangle\in\mathcal{F}} \log\{1 + \exp(-f(h, r, t))\} + \sum_{\langle h',r',t'\rangle\in\mathcal{F}'} \log\{1 + \exp(f(h', r', t'))\} \tag{3.3}$$



(a) $y = \log\{1 + \exp(-f(h, r, t))\}$ (b) $y = \log\{1 + \exp(f(h', r', t'))\}$

Figure 3.4: Logistic loss for a true fact (a) and a false triple (b).

These two losses are the simplest ones and also the first ones used. There is however a wide variety of more complicated losses, generally aiming at enforcing some constraints on the embeddings for logical reasons [87] or to limit overfitting [95]. In order to avoid overfitting, simpler models normalize the handled vectors at each step of the optimization process using the Euclidean distance.

Training is then done from a random initialization state of the parameters by incrementally changing the parameters in order to minimize the loss function by gradient descent: the gradient of the loss with respect to the parameters of the model is computed and then the parameters are updated in the direction opposite to the gradient vector. This is the elementary principle of gradient optimization (cf. Algorithm 3) for tensor models but the methods used in practice can be more complex. An important remark is that usual neural network models are also tensor-based. This means that all the techniques developed to improve training of neural networks are likely to apply as is to the training of KG embedding models. More details on the optimization of tensor-based models can be found in Appendix A, including a short introduction to the Adam optimizer [65], which is used recurrently in the thesis.

Algorithm 3: Training a model by gradient descent.

Input: f, scoring function
Input: γ, step size
Output: P, parameters of the model

randomly initialize P
for a fixed number of epochs **do**
  $\mathcal{L} \leftarrow 0$
  for all facts $\langle h, r, t\rangle$ in training set **do**
    $\langle h', r, t'\rangle \leftarrow \mathrm{NS}(\langle h, r, t\rangle)$
    $\mathcal{L} \leftarrow \mathcal{L} + \ell\left[f(h, r, t), f(h', r, t')\right]$
  $P \leftarrow P + \gamma \nabla \mathcal{L}$
return P

3.2.4 Evaluation techniques

The evaluation of KG embedding techniques comes down to measuring how well the model's scoring function estimates the likelihood of triples to be true. This is done mainly through three tasks: link-prediction, relation-prediction and triple classification. They are all applied under the CWA.

It is crucial to apply these evaluation protocols very efficiently, in order to be able to do it as often as possible. For example, frequent evaluation during the training process

allows a better selection of the hyper-parameters. Chapter 4 presents TorchKGE, as a Python library with a high-performance evaluation module.

Triple Classification

First proposed by Socher et al. in [111], this technique matches the binary classification task on which the whole idea of a scoring functions was formed. Models are evaluated by measuring how accurately they can classify a given triple $\langle h, r, t \rangle$ between true and false. Most of the time, only known facts are used during evaluation. The trained model is used to compute the scores associated to each triple of a test set. If the score is higher than a given threshold, the triple is classified as true. Eventually, the metric reported is the accuracy, that is the share of facts that were correctly classified. This relies on some thresholds that are specific to each relation of the KG and are learned using an evaluation set (distinct from the training and test sets) so that any true fact of the evaluation set cannot have a score inferior to the threshold.

Link-Prediction

First proposed by Bordes et al. in [17], this protocol measures how well the model can complete facts that are missing entities. For each fact $\langle h, r, t \rangle$ in a given test set, two tests are done: a head test to complete the triple $\langle \_, r, t \rangle$ and a tail test to complete $\langle h, r, \_ \rangle$. All known entities e are ranked in both tests by decreasing order of score for the triple $\langle e, r, t \rangle$ (resp. $\langle h, r, e \rangle$). The ranks of the true entities h and t are respectively called the head and tail recovery ranks. Three standard metrics can then be reported: mean rank (MR), mean reciprocal rank (MRR) and hit-at-k (Hit@k).

- MR (mean rank): average of recovery ranks over all facts and both head and tail tests.
- MRR (mean reciprocal rank): average of the inverse of the recovery ranks. This gives a value in $[0, 1]$ and the closer to 1 the better. The interest of this metric compared to MR is that it is less dependent on the number of entities of the KG.
- Hit@k (hit at k): percentage of tests for which the true entity ranks in the top-k entities.

These metrics can also be reported in their *filtered* version, meaning that the recovery ranks taken into account are adjusted by removing all entities with better ranks than the true one that give true facts as well. For example, in the tail test generated with the fact $\langle AlbertCamus, wrote, LaPeste \rangle$, if *La Peste* ranks 5 but among the four top entities there are three books by *Albert Camus* (three known facts), then the filtered recovery rank will be 2. The normal version is then called the *raw* metric. The intuition behind the filtered setting is that the model should not be penalized if it predicts true facts that are simply more likely than the one at hand in the current test.

Some authors refer to this task as relation-prediction, cf. [91] for instance. We make a clear distinction between link-prediction (head or tail entity unknown) and relation-prediction (relation unknown).

Relation-Prediction

This evaluation technique is very close to the previous one and yet it is not very common in the literature. A proposed definition comes in [20]. For each fact $\langle h, r, t \rangle$ of a given test set of facts, a relation test is done to complete the missing relation in $\langle h, \_, t \rangle$. All known relations are ranked by decreasing order of the score of the resulting triple and the rank of the true relation is recorded as *recovery rank*. As in link-prediction, MR, MRR and Hit@k can then be reported in both raw and filtered settings.

3.2.5 Common datasets

There is a variety of datasets in the KG embedding literature. However, a handful of them are very recurrent in the proposed evaluation procedures. Here is a non-exhaustive list focusing on the most common ones:

- FB15k: first introduced by Bordes et al. in [17], this is a subset of Freebase [14] containing roughly fifteen thousand entities selected based on the number of citations in the original dataset. After being the most used dataset for several years, its relevance has seriously been questioned in 2015 by Toutanova et al. who pointed in out some serious data leakage [119]. Many facts of the test set were present in the training set but involving the inverse relation.
- FB15k237: proposed by Toutanova et al. in [119], this is a subset of FB15k in which reverse facts between training and test sets were removed.
- WN18: also introduced by Bordes et al. in [17], this is a subset of Wordnet involving 18 relations between synsets considered as entities. Similarly to FB15k, some data leakage issues were highlighted in WN18 by Dettmers et al. [35].
- WN18RR: proposed y Dettmers et al. in [35], this is a subset of WN18 created by removing data leakage through reverse facts.
- YAGO3-10: proposed by Dettmers et al. in [35], this is a subset of YAGO3 [83] mainly including facts describing people.

Some detailed figures about those datasets can be found in Table 3.1.

Dataset	Entities	Relations	Training facts	Testing facts	Validation facts
FB15k	14,951	1,345	483,142	50,000	59,071
FB15k237	14,541	237	272,115	17,535	20,466
WN18	40,943	18	141,442	5,000	5,000
WN18RR	40,943	11	86,835	3,034	3,134
YAGO3-10	123,182	37	1,079,040	5,000	5,000

Table 3.1: Descriptive figures of the most usual datasets.

3.3 Existing embedding models

As explained earlier, embedding models are divided in groups depending on the form of their scoring function, and subsequently depending on their underlying modelization of the interactions between entities and relations. There are mainly three groups which will be detailed in this section: translational models embedding relation as linear translation between two entities, bilinear models embedding relations as bilinear forms applied to the two entities and eventually deep models. Those latter are usually much less interpretable geometrically. Table 3.4 summarizes the performances of various models with modern re-implementation when available.

3.3.1 Translational models

Translational models, sometimes called linear models, are certainly the most natural and visual ones as they simply try to model relations as translations from one entity to the other in the embedding space or in a sub-space of it. Their scoring functions are based on a distance measure. Empirically, the margin loss seems to give better results than the logistic one when training translational models.

TransE

Bordes et al. presented in 2013 [17] the first model to clearly represent relations as translations. It simply embeds both entities and relations in the same space and defines the scoring function as $f\colon (h,r,t) \mapsto -||\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}||$. Intuitively, this scoring function, if properly fitted to the data, should enforce arithmetic equations of the form $\boldsymbol{h}+\boldsymbol{r} \approx \boldsymbol{t}$ for true facts $\langle h,r,t\rangle$ (as illustrated in Figure 3.5).

It has later been noted by Wang et al. [131], that this simple model lacks expressive power, especially for reflexive, one-to-many and many-to-one relations. First, reflexive relations tend to have an embedding close to 0: for example if $\langle a, \textit{married to}, b\rangle$ is true then $\langle b, \textit{married to}, a\rangle$ is true as well and then both $||\boldsymbol{a}+\boldsymbol{r}-\boldsymbol{b}||$ and $||\boldsymbol{b}+\boldsymbol{r}-\boldsymbol{a}||$ ought to be minimized (where $\boldsymbol{r}$ is the embedding of *isMarriedTo*). This happens simultaneously if and only if $\boldsymbol{r}$ is close to 0 and $\boldsymbol{a} \approx \boldsymbol{b}$. When it comes to one-to-many (resp. many-to-one) relations, the scoring function of TransE tends to embed the many tail (resp. head) possibilities of those relations at the same place. To cope with these limitations, Wang et al. proposed a new model called TransH.

TransH

In 2014, Wang et al. [131] explained that the aforementioned limitations of TransE come from the fact that in TransE "*representations of entities are the same when involved in any relation*". They then proposed to represent relations by a vector and a hyperplane on which entities are projected before the translation is done (as illustrated in Figure 3.5). This gives a scoring function of the form of (3.4) where p_r is the orthogonal projection on the hyperplane specific to the relation r. This hyperplane is defined by its orthonormal vector, which becomes a new parameter of the model to be trained by gradient descent.

$$f\colon (h,r,t) \mapsto -||p_r(\boldsymbol{h})+\boldsymbol{r}-p_r(\boldsymbol{t})|| \tag{3.4}$$

This seems to solve the limitations of TransE because a reflexive relation r can have an embedding vector $\boldsymbol{r}$ close to 0, all information being contained in the projection p_r (i.e. in the vector defining the hyperplane specific to r) and $p_r(\boldsymbol{a}) \approx p_r(\boldsymbol{b})$ is possible without having $\boldsymbol{a} \approx \boldsymbol{b}$.

TransR and TransD

TransH paved the way for a series of algorithms such as TransR and TransD. Each of those model keeps the scoring function of the form of (3.4) but proposes a new way of projecting entities in relation-specific subspaces. In TransR [78], Lin et al. proposed to allow projections to any subspace of a given dimension (which is a hyper-parameter of the model). It improved the expressiveness but increased the number of parameters of the model beyond reason. Indeed, defining a projection to any subspace of dimension k requires a matrix of $d \times k$ parameters (compared to the d parameters of TransH). In TransD [60], Ji et al. proposed in turn to allow any projection defined by a low-rank projection matrix, then limiting the number of parameters.

TorusE

Models presented up to now represent entities as vectors in $\mathbb{R}^d$, others will be presented later embedding in $\mathbb{C}^d$. In 2018, Ebisu et al. explained in [41] that regularization in existing models, which is done using normalization at each step of optimization, prevents the embeddings from behaving as expected. They illustrate how normalized vectors in TransE cannot verify the equation $\boldsymbol{h}+\boldsymbol{r} \approx \boldsymbol{t}$ on which the scoring function is defined.

The authors then proposed a new method called TorusE changing the embedding space to a torus T^d in $\mathbb{R}^d$ (cf. Figure 3.6). Defining for any $(\boldsymbol{x},\boldsymbol{y}) \in \mathbb{R}^d$ the equivalence relation $\boldsymbol{x} \sim \boldsymbol{y}$ if and only if $\boldsymbol{y}-\boldsymbol{x} \in \mathbb{Z}^d$, T^d is then the quotient space

(a) TransE (b) TransH

Figure 3.5: Visualizations of the TransE (a) and TransH (b) models. These figures are extracted from the original paper by Wang et al. [131].

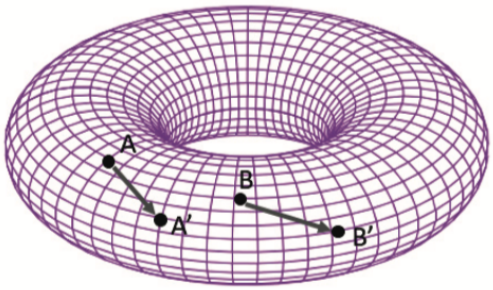


Figure 3.6: Embedding of two facts (A, r, A') and (B, r, B') on a torus T^2. On the torus, $\boldsymbol{A'} - \boldsymbol{A} = \boldsymbol{B'} - \boldsymbol{B}$. This figure is extracted from the original paper by Ebisu et al. [41].

$T^d = \{\{\boldsymbol{y} \in \mathbb{R}^d \mid \boldsymbol{y} \sim \boldsymbol{x}\} \mid x \in \mathbb{R}^d\}$. The authors claim that embedding in T^d is self-regularizing (no normalization needed) and allows vectors to truly verify the arithmetic equations of TransE thanks to a distorted geometry (cf. Figure 3.6). They also claim that on top of performing well, their model is faster to train because of the lack of regularization.

3.3.2 Bilinear models

Bilinear models, sometimes called semantic matching models, use the different approach of modelling relations as bilinear forms (i.e. matrices belonging to $\mathbb{R}^{d \times d}$) that define similarity functions measuring how similar or related two embedded entities are for a given relation. These models have scoring functions of the form of (3.5).

An intuitive interpretation of bilinear models is that they try to learn a geometry specific to each relation by representing it by a bilinear form. If the bilinear forms representing the relations were sufficiently constrained, namely symmetric and positive definite, they would become scalar-products and define relation-specific metrics and subsequently geometries. Now, even if those bilinear forms are not sufficiently constrained, they still act as some relation-specific distortion of the embedding space in which the representations of the entities live.

Empirically, the logistic loss seems to give better results than the margin loss in training bilinear models.

RESCAL

In 2011, Nickel et al. presented in [95] the first bilinear model in its simplest form. Relations are represented as matrices $M_r \in \mathbb{R}^{d\times d}$ and the scoring function is defined as:

$$f\colon (h,r,t) \mapsto \boldsymbol{h}^T \cdot M_r \cdot \boldsymbol{t} \tag{3.5}$$

This model is mainly limited by its high number of parameters, making it hard to train because prone to overfitting. Several models were introduced in the following years to cope with these limitations by proposing new ways of choosing the relation-specific bilinear forms.

DistMult

In 2014, Yang et al. proposed in [141] to limit the choice of bilinear forms to those represented by diagonal matrices. However, as noted in [80], this model is limited to model only symmetric relations. Indeed if M_r is a diagonal matrices, it is also symmetric and then $f(h,r,t) = \boldsymbol{h}^T \cdot M_r \cdot \boldsymbol{t} = \boldsymbol{t}^T \cdot M_r^T \cdot \boldsymbol{h} = \boldsymbol{t}^T \cdot M_r \cdot \boldsymbol{h} = f(t,r,h)$.

This model can also be seen as an implementation of the CP decomposition for three-dimensional tensors. For an adjacency tensor $\mathcal{T} \in \mathbb{R}^{n\times n\times m}$, the d-rank CP decomposition results in three matrices $(U,V,W) \in (\mathbb{R}^{n\times d} \times \mathbb{R}^{n\times d} \times \mathbb{R}^{m\times d})$ that can be used as embeddings of the head entities U and tail entities V and embeddings of the relations W. This is what the authors of DistMult proposed, adding the constraint $U = V$.

HolE

In 2016, Nickel et al. proposed in [94] a new model of *holographic embedding* that uses circular matrices as in (3.6). The number of parameters in the model is still the same as in DistMult but it can now model other relations than the symmetric ones.

$$M_r = \begin{pmatrix} r_1 & r_d & \dots & r_2 \\ r_2 & r_1 & & r_3 \\ \vdots & \vdots & & \vdots \\ r_d & r_{d-1} & \dots & r_1 \end{pmatrix} \tag{3.6}$$

ComplEx

In 2016, Trouillon et al. released in [122] a new way of representing all kinds of relations with an embedding in complex numbers. Entities are embedded in $\mathbb{C}^d$ and relations are represented by diagonal bilinear forms $M_r = \mathrm{diag}(\boldsymbol{r})$ with $\boldsymbol{r} \in \mathbb{C}^d$. The scoring function is defined as the real part of the Hermitian product $f\colon (h,r,t) \mapsto \mathrm{Re}(\boldsymbol{h}^T \cdot M_r \cdot \bar{\boldsymbol{t}})$. In their paper, the authors justify the use of complex embeddings by linking the matrix factorization problem of the adjacency matrix of a relation to its spectral decomposition that is only possible in $\mathbb{C}$ for some non-symmetric matrices.

Like with DistMult, ComplEx can be seen as an implementation of the CP decomposition of the adjacency tensor but this time the constraint is that $U = \bar{V}$.

ANALOGY

In 2017, Liu et al. presented a new algorithm in [80] called ANALOGY whose goal was to model analogical structures found in data. Analogical structures are famously illustrated by the natural language processing example of Mikolov et al. [86]: *man* is to *king* as *woman* is to *queen*. It is naturally extended to KGs as the implicit links *beingCrowned* and *feminineOf* of this example are possible relations of a KG.

In their paper, the authors argue that this analogical structure can be achieved with embeddings by enforcing commutativity of the relation-specific bilinear forms. They

then show that constraining any pair of relations to be commutative can be simplified to constraining the relation-specific bilinear forms to almost-diagonal matrices. This final constraint is compatible with a gradient-descent optimization process.

Definition 25 (Almost-diagonal matrix)**.** *An almost diagonal matrix is a matrix* $M \in \mathbb{R}^{d \times d}$ *such that any diagonal block is either a scalar or a* 2×2 *block of the form* $\begin{pmatrix} x & y \\ -y & x \end{pmatrix}$*. All other elements of* M *should be zero. The set of* $d \times d$ *almost-diagonal matrices with* k *scalar elements is denoted as* $\mathcal{B}_d^k$*.*

Eventually, they state that in their implementation of ANALOGY, relation-specific matrices are forced to live in $B_d^{d/2}$. This new model still has the same number of parameters as DistMult, which is lower than RESCAL.

3.3.3 Deep models

The translational and bilinear models previously presented are based on differentiable tensor computations and can thus be considered as shallow neural networks [116]. Models with deeper architectures have also been proposed with additional hidden layers used to define more complex scoring functions. Those architectures differ from one model to the other and exhaustively listing them would be of poor interest. Instead the current section explains the underlying structure common to all the models.

In order to compute the score of a fact $\langle h, r, t \rangle$, deep models usually have an architecture divided in two parts. The first part projects h, t and r to corresponding embeddings $\boldsymbol{h}, \boldsymbol{t}$ and $\boldsymbol{r}$ in $\mathbb{R}^d$ and processes them together using linear transformations such as concatenation, reshaping, matrix and vector products along with coordinate-wise non linear functions. The output of the first part of the network is a latent representation of the input fact. The second part of the network uses the latter to compute the score. The architectures of both parts define the specificity of the model and a variety of neural network layers can be involved: fully connected [39], convolutional [108, 35, 93] or recurrent [147].

The embeddings of entities and relations are usually not the only parameters of the model as the parameters of the hidden layers (used to compute the fact representation and the final score) need to be trained as well. Those are however shared by all entities and relations.

In 2013, Bordes et al. proposed the first deep architecture called Semantic-Matching Energy (SME), later published in [16]. Two versions of SME were presented: linear and bilinear that are respectively comparable to translational and bilinear models. Later in 2013, Socher et al. proposed in [111] a model called Neural Tensor Network (NTN) introducing the use of coordinate-wise non-linear functions. Then in 2014, Dong et al. proposed a simpler approach called Multi-Layer Perceptron (MLP) [39].

More recently, in 2017 and 2018, models including convolutional layers in the second part of their structures were introduced: ConvE by Dettmers et al. and ConvKB by Nguyen et al. [35, 93]. They process the concatenations of embeddings as images and use convolutional layer to extract relevant features from the weights. A visualization of ConvKB is available in Figure 3.7. Such layers, which proved very successful in image processing are parameter-efficient and seem to increase the expressive power of models.

Simultaneously, Graph Convolutional Networks (GCNs) have become a cornerstone of the ML literature on graphs. First introduced in 2014 [24] and in 2016 [67, 34], these neural networks feature aggregation mechanisms over graphs in order to propagate features from a node to its neighbors, like a message passing algorithm. These will be more detailed in Chapter 6. Several adaptations of GCNs to KGs have been proposed, the first of them being R-GCN by Schlichtkrull et al. in 2018 [108]. As it lacked an explicit representation of relations, it was extended a year later by Ye et al. in [143].

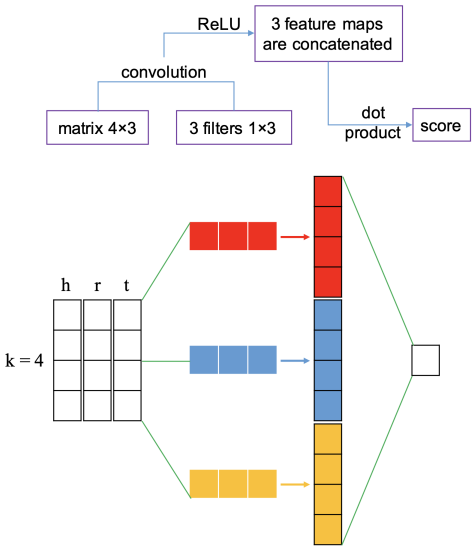


Figure 3.7: Processing of a fact $\langle h, r, t \rangle$ by ConvKB. The embedding size is $k = 4$, three convolution filters are used and the activation function g is ReLU. (Source [93])

Eventually, Vashishth et al. proposed CompGCN in 2019 [125] which outperforms most of the existing deep models in link-prediction.

3.3.4 Comments on the available literature

In spite of the amount of published research and the performances reported, a rigorous experimenter might be disappointed by the results of state-of-the-art models when applied to *real-world* data, that is data that has not been selected, designed and processed specifically for this research topic. This has been reported in the literature with several studies taking a critical look on the reported performances and implying that a large amount of the performance gain reported by some authors might actually come from poorly designed evaluation processes.

Toutanova et al. [119] along with Dettmers et al. [35] have already been cited in Section 3.2.5 for exhibiting data leakage in the two most widespread datasets at the time, FB15k in 2015 and WN18 in 2018. This caused a substantial drop in performances of existing models. For example, the filtered Hit@10 of ConvE went from 85.6% on FB15k to 48.1% on FB15k237 and from 95.5% on WN18 to 50.4% on WN18RR [35]. Despite the publication of corrected versions, namely FB15k237 and WN18RR, many papers still report performances on the original flawed datasets (cf. [38] for example).

More recently, Akrami et al. proposed in 2020 an eye-opening analysis of data redundancy in common datasets [2]. Three possible causes were highlighted:

- inverse relations (cf. Definition 10).
- duplicate relations (e.g. *hasCitizenship* and *hasNationality*).
- Cartesian product relations (i.e. relations that are true between any pair of a subset of $\mathcal{E} \times \mathcal{E}$).

These relations are particularly problematic when they cause redundancy shared across training and test facts. The authors then proceeded to measure precisely the extent of the redundancy in FB15k and WN18 and formulate some remarks on remaining flaws in the corrected versions. For example, WN18RR still contains symmetric relations with up to 34% of the training triples involving a single symmetric relation. When it comes

to FB15k237, Akrami et al. regret that the reverse property available in Freebase was not used.

Another recurrent flaw in the KG embedding literature was pointed out by Sun et al. in 2020 [117]. The authors noticed that performance gains claimed by some articles can actually be credited to a flawed evaluation method. Precisely, when entities are being scored for link-prediction, several recent models give the exact same score to many entities. This makes it difficult to break ties and many articles actually use the most advantageous strategy to do so.

Another simpler criticism of the evaluation strategy applies to the use of fixed train, test and validation splits. Though it is understandably motivated by the high computation cost of applying a model selection technique such as k-fold cross validation, it undoubtedly can lead to a whole field of research iteratively overfitting the data, article after article.

Eventually let us cite [63] and [105] who wisely explained the successive progress of the proposed models by the simultaneous progress in optimization and training methods. To do so, they re-implemented, re-trained and re-evaluated many models including the older ones. The results were that training strategies (optimizer, parameter regularization, learning rate choice) play a major role in the gain of performance observed in recent literature. The authors managed to train older models such as RESCAL up to outperforming recent and more complex methods such as ConvE. The results of their study are partially reported in Table 3.3.

	Model	Number of parameters
Translational Models	TransE	$\mathcal{O}(n_e d + n_r d)$
	TransH	$\mathcal{O}(n_e d + n_r d)$
	TransR	$\mathcal{O}(n_e d + n_r dk)$
	TransD	$\mathcal{O}(n_e d + n_r k)$
	TorusE	$\mathcal{O}(n_e d + n_r d)$
Bilinear Models	RESCAL	$\mathcal{O}(n_e d + n_r d^2)$
	DistMult	$\mathcal{O}(n_e d + n_r d)$
	HolE	$\mathcal{O}(n_e d + n_r d)$
	ComplEx	$\mathcal{O}(n_e d + n_r d)$
	ANALOGY	$\mathcal{O}(n_e d + n_r d)$
Deep Models	SME	$\mathcal{O}((n_e + n_r + k)d)$
	ConvE	$\mathcal{O}(n_e d + n_r k)$

Table 3.2: Number of parameters (space complexity) of models presented in Section 3.3. n_e is the number of entities, n_r the number of relations, d the embedding dimension, k can either be the relation embedding dimension or the latent space dimension (this is optional).

		FB15k237		WN18RR	
		MRR	Hit@10	MRR	Hit@10
First reported results	RESCAL [95]	0.270	42.7	0.420	44.7
	TransE [17]	0.294	46.5	0.226	50.1
	DistMult [141]	0.241	41.9	0.430	49.0
	ComplEx [122]	0.247	42.8	0.440	51.0
	ConvE [35]	0.325	50.1	0.430	52.0
Recent re-implementations [105]	RESCAL	0.356	53.6	0.468	52.1
	TransE	0.303	48.8	0.227	52.6
	DistMult	0.342	52.3	0.452	53.1
	ComplEx	0.351	53.7	0.477	54.3
	ConvE	0.337	52.8	0.447	50.8

Table 3.3: Performances of various embedding models as first reported in the original articles and as reported by [105]. The table itself is from [105].

3.4 Reasoning with knowledge bases

The previous section introduced ML methods for KG completion using embeddings. Historically however, the first solutions to the problem were symbolic and came from the field of semantic reasoning. The current section introduces it along with classical algorithms and eventually explains how it can help machine learning methods.

This section partially covers the following article:

Combining Embeddings and Rules for Fact Prediction.
Boschin, A., Jain, N., Keretchashvili,G., Suchanek,F. (2022)
Postprint presented at the *International Research School in Artificial Intelligence* (Bergen, Sweden, June 2022).
Reference [22]
Personal contribution: parts of Section 3 (Embedding models), parts of Section 4 (Embedding Methods with Logical Components) and Section 5 (Rule Mining with embedding techniques).

3.4.1 Rules at the core of semantic reasoning

Semantic reasoning is the act of inferring new facts by exploiting the ones recorded in a KB along with the accompanying ontology (taxonomy and axioms). Chapter 2 stated that KBs come with axioms that are usually manually defined and allow no exceptions. They are enforced during the KB construction.

Constraints, on the other hand, come from the data itself and can have some exceptions. For example, is it possible to learn that the relation *marriedTo* is usually symmetric, exceptions coming from missing facts. Such patterns in the KB are formalized by Horn rules, which will be defined now.

Definition 26 (Atom)**.** *An* atom *is an expression of the form* $\langle \alpha, r, \beta \rangle$*, where* r *is a relation and* α*,* β *are either entities or variables [73]. Variables are usually written in lower case and entities in upper case.*

A *substitution* σ is a partial mapping from variables to entities. For example, $\sigma = \{x \rightarrow \textit{Freddie Mercury}\}$ is a substitution and $A = \langle x, occupation, singer \rangle$ is an atom with one variable x. $\sigma(A)$ yields the fact $\langle FreddieMercury, occupation, singer \rangle$ known to be true.

An atom is said to be *instantiated* if at least one of its arguments is an entity. If both arguments are entities, the atom is *grounded* and becomes a triple. A *conjunction* of atoms $B_1, \ldots, B_n$ is the logical `AND` concatenation of the atoms, noted $B_1 \wedge ... \wedge B_n$. For example, the conjunction $\langle x, occupation, singer \rangle \wedge \langle x, birth\ place, Tanzania \rangle$ intuitively designates all entities who were born in Tanzania and who are professional singers.

Definition 27 (Horn Rule)**.** *A Horn* rule *is a formula of the form* $B_1 \wedge ... \wedge B_n \Rightarrow H$*, where* $B_1 \wedge ... \wedge B_n$ *is a conjunction of* body atoms*, and* H *is the* head atom*.*

A rule is *grounded* if all of its atoms are grounded and it is *closed* if every variable in the head appears in at least one body atom. Two atoms A, A' are *connected* if they have common variables. It is common [49, 48, 97] to impose that all atoms in a rule are transitively connected and that rules are closed.

To conclude, rules are an expression of data constraints, just like axioms, but they usually reflect patterns that must be mined from the facts. As opposed to axioms, rules allow exceptions: the fewer the better the quality of the rule. Two usual metrics for rule quality are the *support* (the number of positive examples predicted by the rule) and the *confidence* (the proportion of the triples predicted that are known to be true). Rules are sometimes called *soft* as opposed to *hard* axioms.

3.4.2 Rule mining for knowledge base completion

Given a rule $R = B_1 \wedge ... \wedge B_n \Rightarrow H$ and a substitution σ, we can apply σ to both the body and the head and obtain an *instantiation* of R denoted $\sigma(R)$. An example for a rule is $\langle x, spouse, y\rangle \wedge \langle x, residence, z\rangle \Rightarrow \langle y, residence, z\rangle$, noted R^*. It could be instantiated by $\sigma = \{x \rightarrow Alice, y \rightarrow Bob, z \rightarrow Paris\}$, and obtain $\sigma(R^*)$ as $\langle Alice, spouse, Bob\rangle \wedge \langle Alice, residence, Paris\rangle \Rightarrow \langle Bob, residence, Paris\rangle$.

Definition 28 (Prediction)**.** *Given $\mathcal{K}$ a KB, σ a substitution and R a rule with head atom H and body atomes $(B_i)_{i\in\{1,...,n\}}$, if $\sigma(B_i) \in \mathcal{KB}\ \forall i \in \{1, ..., n\}$, we call $\sigma(H)$ a prediction of R from $\mathcal{KB}$.*

The previous example rule R^* yields the prediction $\langle Bob, residence, Paris\rangle$ if the facts $\langle Alice, spouse, Bob\rangle$ and $\langle Alice, residence, Paris\rangle$ are known in the KB.

KB completion by semantic reasoning subsequently comes down to mining rules automatically and instantiating as many of them with the known facts, thus yielding predictions. A restriction compared to the ML approach is that it is not possible to complete any atom like in link-prediction but only the ones that exist in the head of a rule.

The key step is the automatic rule mining that is done by Inductive Logic Programming (ILP). A precise definition of ILP is given in [116]. It typically takes as input a set of *positive examples* (i.e. facts that the rules shall predict), and a set of *negative examples* (facts that the rules must not predict). ILP faces several challenges: first, as explained previously, KBs usually do not record negative examples. Methods to generate false triples have already been discussed in Section 3.2.2. Another challenge is that a strict application of the definition of ILP to rule mining would find only rules that do not tolerate exception (*hard* rules). A solution is for rule mining to aim for rules that have high support and confidence.

AMIE [49] was one of the first rule mining systems for large KBs under the Open World Assumption. It is a greedy method that basically explores the set of all possible rules and only keeps the ones respecting confidence and support constraints. It starts with the most general rules (such as "*everybody is married with each other*") and refines them until their confidence is high enough but stops once the support gets too low. This relies on the observation that the support of a rule decreases monotonically when a rule is made more specific. The latest version, called AMIE3 [73], remains one of the most used methods for rule mining. It dramatically improves the computation time by improving the exploration strategy and implementing some coding tricks.

Many rule mining systems have been proposed subsequently to AMIE. Two notable examples are RuDiK, which can mine negative rules [97], and AnyBURL, which has successfully been applied to link-prediction (cf. Table 3.4). Another interesting method is DRUM that formulates rule mining as a linear optimization problem solved by gradient descent.

Eventually, let us remark that it is also possible to design symbolic methods that do not rely on rules and reasoning. For example, a symbolic method that can be applied to link-prediction is Concepts of Nearest Neighbors (CNNs) [45]. It uses recurrent KG neighboring patterns to define a graph metric that can then be used for classification by nearest neighbors.

3.4.3 Embedding methods with logical components

A healthy practice when engineering ML models is to incorporate as much prior knowledge about the data in the model definition. The extreme approach of designing a model with a huge expressive power by including a massive number of parameters and hope for the model to learn by itself everything there is to learn about the data is indeed usually

inefficient. Large language models such as BERT [36] or GPT-3 [23] show how costly designing and training such models can become.

The previous section recalls that KBs usually come with hard constraints on the entities and relations in the form of axioms and explains that soft rules can be learned directly from the KB and be used to predict some missing facts.

Surprisingly, axioms which are usually available off-the-shelf were originally left untapped by ML techniques. The trend at the beginning of the 2010s was to let ML models with many parameters solve any problem and in particular KG completion. When Toutanova et al. spotted some flaws in the (too) widely used dataset FB15k [119] in 2015, disappointment spread with the drop in performances. As explained in Section 3.3.4, in 2020 Akrami et al. studied the role of data redundancy in artificially boosted performances [2]. They also showed that it was possible to beat the performances of complex embedding models with simple rule mining approach when guided by reasoning.

There are several arguments in favor hybrid methods for automatic KG completion and both axioms and soft rules seem to be good allies of KG embedding methods. First, rules and axioms constitute logical expressions of explainable patterns that can usually be easily integrated in model designs. Moreover, rule mining is usually designed, applied and evaluated under the OWA, to predict rules that are not yet in the KB. This is not the case of ML as it needs to be trained under the CWA, thus causing the paradox already explained in Section 3.2.2. Eventually and most importantly, the empiric results show that semantic reasoning improves the performance of ML models.

There are mainly three ways semantic reasoning has been integrated in ML approaches. The simplest one is to force or at least nudge the embeddings and scoring function to comply with axioms and rules. The simplest constraints to implement apply to relations (e.g. equivalence, inverse, symmetry and subsumption) because they can be directly enforced on the embedding vectors. For example, [87] proposed in 2017 a method to implement equivalence and inverse axioms by intuitively adapting them to each model: for TransE, if r_1 and r_2 are to equivalent relations then their embeddings should be equal. If the constraint comes from a mined rule, then the model should be incentivized to give close embedding to the two by adding to the loss function a regularization term of the form $\lambda \times ||\boldsymbol{r}_1 - \boldsymbol{r}_2||$. [38] has the same approach and enforces subsumption constraint on ComplEx. If the constraint comes from a mined rule, the authors propose to include the rule confidence in the process. [43] proposes in addition to enforce taxonomic compliance by formalizing typing constraint as a particular case of relation subsumption. The authors of those three articles claim that hybridization caused sensible performance improvement on link-prediction.

Another possibility is to use axioms to enrich the set of training facts on which the embedding models are trained. In 2021, Amato et al. proposed to use ontological axioms such as taxonomic sub-typing and equivalence or reverse of relations to generate more training facts [40]. They also propose a negative sampling strategy that corrupts existing facts by violating taxonomic axioms, making sure that the new samples are not in the KB. Also in 2021, Jain et al. proposed to improve negative sampling with reasoning with a framework called ReasonKGE to train embedding models [59]. The idea is to use predicted triples that do not comply with the taxonomy as new negative samples (cf. Figure 3.8). The authors claim that the framework improves the performance of all tested models in link-prediction by up to 10% in Hit@10. Unfortunately, the only common dataset on which ReasonKGE tested is YAGO3-10, comparative results are reported in Table 3.4.

Eventually, some authors attempt to synchronize rule mining and KG embedding mainly by using the scoring function to score rules. For example in 2016, Guo et al. use t-norm logic to score rules mined by AMIE with the scoring function of an embedding model [53]. The same research group proposed in 2018 a more comprehensive method

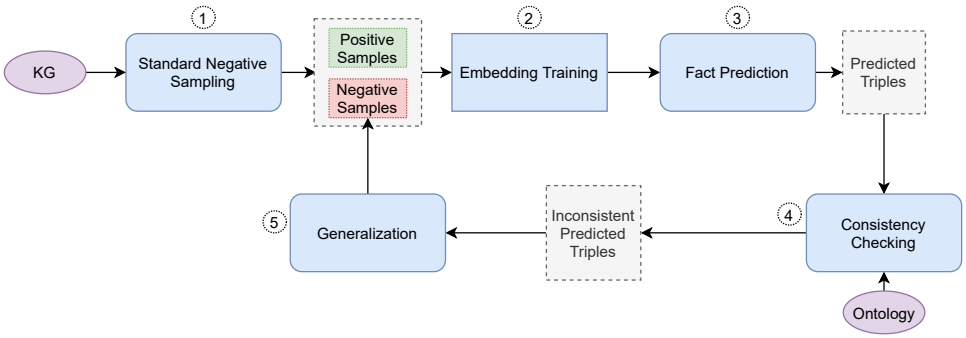


Figure 3.8: Schematic representation of the ReasonKGE framework [59].

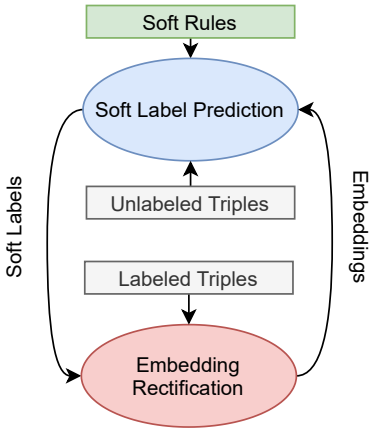


Figure 3.9: Schematic representation of the RUGE framework [54].

called RUGE [54] that alternatively mine rules on a dataset enriched by the embedding model and train the embedding model on a dataset enriched with the mined rules, while score triples generated by rule mining with the scoring function of the embedding model.

Eventually, note that there are very few works trying to improve rule mining with ML methods. A selection of them were presented in [22].

3.5 Conclusion

Several lessons can be learned from the large variety of models previously reviewed. Some performance metrics are reported in Table 3.4.

First, many factors can influence the performances of a model. An obvious one, and maybe the most important one, is the dataset. No model yet has been able to surpass all the others on all the datasets. Though ComplEx and CompGCN seem to be better off than most of the models, the score differences are not that impressive relatively to their absolute value. For example on FB15k237, CompGCN has a 53.5% Hit@10 score, which is almost 4% better than the simplest model TransE. Considering that models do not rank the true entities in the top ten ones approximately half of the time, this slight improvement is not really significant. The cost of training deep-learning models is not obvious.

Another intuition relative to datasets and coming from Table 3.4 is that the performances of most models might not generalize well to real-world data. An argument is that YAGO3-10 is much less used than FB15k237 and WN18RR in the literature making it a possible proxy estimator for real-world performances. The results show that most models perform poorly comparatively to ComplEx or CompGCN and even the simpler TransE. This will be further discussed in Chapter 5 that introduces WDV5, a new dataset extracted from Wikidata.

Second, the performances of most models reported after retraining in [104] confirm the importance of high-end training techniques comparatively to more complex models.

Eventually, it is interesting to note that the reasoning model AnyBURL performs best on WN18RR, while TransE ranks last. It might be explained by the remark formulated in [2] that WN18RR still contains some symmetric relations and that symbolic methods of rule mining are interestingly good at mining those, while TransE represents them poorly. This is in favor of conceiving hybrid methods that take advantage of both semantic reasoning and machine learning.

To conclude, it seems that a reasonable method in practice is to keep the model conception simple, like TransE. Resorting to reasoning to improve training or inference in hybrid-methods also seems fruitful. Importantly, diverse datasets should be used for evaluation and particular attention must be paid to the training procedure. Chapter 4 will introduces TorchKGE, a Python library featuring a very efficient evaluation module that is useful to train models properly.

		FB15k237		WN18RR		YAGO3-10	
		MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
Translational models	TransE [105]	0.310	49.7	0.206	49.5	0.501	67.4
	TorusE [104]	0.281	44.7	0.463	53.4	0.342	47.4
Bilinear models	CP [108]	0.182	35.7	-	-	-	-
	DistMult [104]	0.313	49.0	0.433	50.2	0.501	66.1
	HolE [104]	0.303	47.6	0.432	48.8	0.502	69.2
	ComplEx [104]	0.349	53.0	0.458	52.1	**0.576**	**70.4**
	Analogy [104]	0.202	35.4	0.366	38.0	0.283	45.7
Deep models	ConvE [104]	0.305	47.6	0.427	50.8	0.488	65.8
	ConvKB [104]	0.230	41.5	0.249	52.5	0.420	60.5
	R-GCN [108]	0.248	41.7	-	-	-	-
	VR-GCN [125]	0.248	43.2	0.475	53.7	-	-
	CompGCN [125]	**0.355**	**53.5**	0.479	54.6	-	-
Symbolic models	AMIE+ [45]	0.143	24.1	-	-	-	-
	CNN [45]	0.286	42.9	-	-	-	-
	AnyBURL [104]	0.324	48.9	**0.485**	**56.0**	0.528	66.1
Hybrid models	RUGE [54]	-	-	-	-	0.431	60.3
	ReasonKGE (TransE) [59]	-	-	-	-	0.367	62.9
	ReasonKGE (ComplEx) [59]	-	-	-	-	0.530	66.8

Table 3.4: Performances in filtered setting of various models as reported by the cited sources. [104] proposes a comparative study of models by retraining the proposed models implementations in the same framework.

Chapter 4

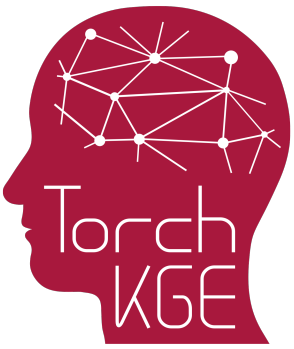
TorchKGE

This chapter covers the following article:

TorchKGE: Knowledge Graph Embedding in Python and PyTorch
Boschin, A. (2020)
Postprint presented at the *International Workshop on Knowledge Graph co-located with the 26th Conference on Knowledge Discovery and Data Mining* (August 2020).
Reference [18]

4.1 Motivations

Chapter 3 introduced a variety of ML models, most of them relying on the definition of a scoring function and on a common KG completion evaluation procedure. It has however been argued that the successive performance gains reported by some authors are questionable and one of the main reasons is related to implementation and optimization frameworks.



A first issue is the heterogeneous model implementations that are likely to make the results incomparable. When authors provide accompanying code, and it is not always the case, a variety of programming languages (e.g. C++, Python, Java) and computing frameworks (e.g. Theano, SciPy, TensorFlow, PyTorch) are used. A second issue, that Section 3.3.4 already raised, is that some recent advances are more likely to be caused by better training strategies than by better models [105]. These two problems call for a unique framework to experiment with all the models. Such a library should provide efficient tools specific to KG completion but it should also be compatible with recent optimization libraries. These are two primary objectives of TorchKGE.

A key feature for an implementation framework is computation efficiency and given that most KG embedding models are built with tensors, parallel computing is a must. Recently Graphics Processing Units (GPUs) have made the access to heavy computation cheaper and cheaper. Such GPUs are usually powered by CUDA for vectorization. First released in 2007 [96] by Nvidia, CUDA indeed quickly became a standard tool for

accelerated computation by being adopted by popular tensor frameworks PyTorch [99] and TensorFlow [1]. These libraries propose off-the-shelf methods to train parametric functions with diverse gradient descent methods, thus making it easy to training KG embedding models with efficiency. Support for one of those is a requirement.

Specific to KG embedding models, the evaluation procedures detailed in Chapter 3 are not directly proposed by the main computing frameworks. Effectively implementing an evaluation module is however crucial because it conditions the ability to better tune the hyper-parameters. Indeed, hyper-parameter tuning usually comes down to testing several value combinations (with more or less sophisticated strategies) and selecting the one yielding the best results. Building such a module that can be applied to any embedding model is however tricky because of the diversity of scoring function possible.

It has become clear that a unique framework for KG embedding could be of key interest to the research community to make the comparison of embedding models easier and more accurate. It could also help the adoption of the most advanced techniques by the industry by proposing off-the-shelf solutions. A performance measure of such a framework is the efficiency of the code in both training and evaluating models.

First released in 2019, TorchKGE, which stands for Knowledge Graph Embedding in PyTorch, aims to meet the challenge by proposing an efficient API for the implementation and evaluation of KG embedding models. It also comes with pre-trained implementations of models and a handful of useful tools to represent KGs in memory and handle models. This open-source library is implemented in Python and relies on PyTorch as computing backend. It is currently in version 0.17.7 and supports Python 3.8, 3.9 and 3.10.

4.2 Other existing libraries

TorchKGE was not the only Python library to propose a framework for KG embedding models and tools for evaluation. Here is a brief review of the other existing frameworks.

4.2.1 OpenKE

OpenKE (for Open-source Knowledge Embedding)[1] was first released in 2017 by a research group from the Tsinghua University in China [55]. Despite being first based on TensorFlow, PyTorch was then adopted as main computing backend in 2020. In search of efficiency, some methods are implemented in C++, which is faster than Python thanks to compilation and static typing among other things. This leads to good performances in model training. The evaluation modules, however, are naively implemented and fail at taking full advantage of GPU acceleration (cf. Section 4.5). A major drawback of OpenKE is that its development does not implement any of the CI/CD practices (cf. Section 4.3 for an extensive definition of CI/CD) preventing the library from being used in any serious or industrial system. For example, it does not number releases. Eventually, let us remark that the project does not seem to be maintained anymore as the last commit on the GitHub repository dates from April 2021.

4.2.2 Ampligraph

Another contender for TorchKGE is AmpliGraph[2]. It was first released in March 2019 by a team from the Accenture Labs based in Dublin [31]. Currently in version 1.4.0, its development follows commons good practices as the development team aims at turning it in a serious industrial tool. For example, they provide an extensive documentation. Eventually, the TensorFlow computing backend yields efficient model training perfor-

[1] https://github.com/thunlp/OpenKE
[2] https://github.com/Accenture/AmpliGraph/

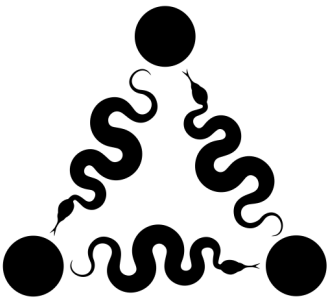


Figure 4.1: Pykeen logo

mances but regrettably the link-prediction evaluation module is not implemented in an efficient enough way.

4.2.3 Pykg2vec

A third python library is Pykg2vec[3], which was first released in March 2019 [144]. Using TensorFlow, this library provides several modules (models, wrappers, visualization), but its development is still at a very early stage and as the current version number suggest (0.0.52), the API is not stable. In the tests presented at this end of the current chapter, we were unable to make version 0.0.50 work with enough satisfaction to compare it to the other libraries.

4.2.4 PyTorch-BigGraph

PyTorch-BigGraph is also worth mentioning. Developed by Facebook AI Research (FAIR) [76], the library aims at embedding massive graphs that cannot fit on single machines. This is however a specific (and very useful) use-case which is different from the one of TorchKGE. It will not be further included in the current chapter.

4.2.5 The need of a new library

In 2019, OpenKE and AmpliGraph seemed to be the two best candidates for providing a simple and efficient framework for KG embedding. However, their implementations of link-prediction evaluation, which is at the core of the development of KG embedding models was really unsatisfactory (cf. Section 4.5 for comprehensive benchmarks). Divergence in the project vision led to the choice of developing a new library from the ground up (TorchKGE), instead of contributing improvements to the existing ones. They will be detailed in the following sections.

4.2.6 Pykeen

Eventually, let us introduce Pykeen[4]. This library was at first not included in our work [18] in 2020 because when the development of TorchKGE started, Pykeen was a discrete package at a very early stage in its development. It was in alpha version 0.0.26 since august 2019 and had no usable documentation. The first official release, which was made in June 2020, was a major step up. Despite its slow take-off, Pykeen has since managed to build-up a large community of contributors, approximately thirty

[3] https://github.com/Sujit-O/pykg2vec
[4] https://github.com/pykeen/pykeen

according to GitHub, making it the current most serious contender to TorchKGE. The two libraries now propose the best performing evaluation modules (cf. Section 4.6). Currently in version 1.9.0, Pykeen proposes a large collection of implemented models, negative sampling methods and other useful tools.

4.3 Conception choices

As previously explained, the goal of TorchKGE is to provide users with a flexible KG data-structure, fast evaluation modules, off-the-shelf model implementations and model interfaces to make it easy to develop new models and compare them to existing state-of-the-art but also to integrate painlessly existing models in other projects.

4.3.1 GPU acceleration

As explained earlier, a key specification of TorchKGE from the start was the computing efficiency, making the support of GPU acceleration a requirement. The two best solutions for accelerated tensor computations are still currently TensorFlow, developed by Google [1], and PyTorch, based on the Torch framework developed by Meta [99]. The choice naturally fell on PyTorch as in 2019, its user experience was way superior to the one of TensorFlow. The latter was greatly improved with the release of TensorFlow 2 by the end of 2019 but not enough to justify the change in TorchKGE. The fact that Ampligraph relied on TensorFlow is the reason why we choose not to contribute to the project and instead to pick-up the development of TorchKGE.

PyTorch is an open-source machine learning library published under BSD license. It provides support of GPU acceleration for tensor computations along with a high-performance auto-differentiation system useful to train models with iterative methods such as gradient descent. Its API is simple to learn and use and yet highly efficient. Another very interesting feature is the seamless integration with the Numpy library [128], which is one of the most widespread tools in machine learning research. Those reasons make PyTorch one of the best choices of backend for TorchKGE.

4.3.2 API design

Inspired by the design of PyTorch, TorchKGE can be used by advanced user to implement new models or to tweak existing ones. It can also be used more simply with one of the pre-implemented models and wrappers.

The design of the library matches the intuitive one of PyTorch. This allows easy adoption by users that are already familiar with engineering ML models in PyTorch. Namely, a requirement for models is to implement a core interface from PyTorch: `torch.nn.module`. This simple class inheritance gives access to all the useful GPU acceleration tools of PyTorch (e.g. automatic differentiation) by a simple call to the object method `cuda` (cf. Figure 4.2). The interface also comes with handy tools for parameters management (training and evaluation modes, model parameters listing, saving and reloading of model state).

Another advantage of matching the design of TorchKGE with the one of PyTorch is that it brings built-in support of the many resources available in PyTorch (e.g. optimization modules) but also resources designed for PyTorch. A useful example is `PyTorch-Ignite`[5], which is described as a high level library for training and evaluating PyTorch models. It includes very important tools such as early stopping and checkpoint management. A code snippet of training a model with early-stopping using Ignite can be found in Figure 4.7 at the end of the chapter. Easy resort to such tools can help solve the training problem raised in the previous chapter.

[5] https://pytorch.org/ignite

```python
from torch import cuda
from torchkge.utils.pretrained_models import load_pretrained_transe
from torchkge.utils import load_fb15k237

kg_train, kg_eval, kg_test = load_fb15k237()

model = load_pretrained_transe(dataset='fb15k237', emb_dim=150)

if cuda.is_available():
    model.cuda()
```

Figure 4.2: Loading a pre-trained TransE model and moving it to the GPU.

Some wrappers are also already implemented for off-the-shelf use of existing models. See Figure 4.3 for an example of the `trainer` wrapper. Eventually, the library includes as few dependencies as possible to ease both the maintainability on the developer side and the integration on the user side.

4.3.3 Good development practices

From the start, TorchKGE was designed with rigorous development practices in mind. This is the only way the project could lead to a serious library that could meet the objectives presented in Section 4.1.

Documentation

First, an extensive documentation is provided online[6]. It includes class and method descriptions along with code snippets and full example scripts for model definition, training and evaluation. All the included implementations are documented with references to the original papers. Documentation also provides guidelines for community contributions.

Continuous integration

Integrating new code to existing systems can lead to what is usually referred to as *integration hell*. It is the set of problems that can arise when making changes to a piece of code that is already in production (i.e. part of a system with active users). An intuitive example is backward compatibility: modifying a function or an object can break older pieces of code that were using it. The function and method signatures can have changed for example, to add an argument. ML engineers can experience this integration hell at a very early stage of research and development, even if they are still the only users of their code.

There is a set of good practices known as Continuous Integration (CI) that helps developers cope with integration hell. CI relies on the use of a Distributed Version Control System (DVCS), e.g. `git`[7]. It enables teams of developers to *commit* changes to the code and organize them with *branches*. They can work offline and only push modifications occasionally, eventually merging simultaneous contributions with the help of a conflict solver. Code commits should be as frequent as possible and documented to ease the integration. The code is also required to respect the PEP8 style guide in order to ease collaboration [124].

TorchKGE is hosted on the well-known GitHub hosting service, which obviously provides git support along with useful features such as access control, bug tracking, feature requests and task management. Community contributions, which are encouraged, can be submitted as *Pull Requests* that are assigned, reviewed and eventually merged after validation.

[6] https://torchkge.readthedocs.io/en/latest/
[7] https://git-scm.com/

```python
from torch import cuda
from torch.optim import Adam

from torchkge.evaluation import LinkPredictionEvaluator
from torchkge.models import TransEModel
from torchkge.utils.datasets import load_fb15k237
from torchkge.utils import Trainer, MarginLoss


def main(use_cuda):
    # Define some hyper-parameters for training
    emb_dim = 100
    lr = 0.0004
    margin = 0.5
    n_epochs = 1000
    batch_size = 32768

    # Load dataset
    kg_train, kg_val, kg_test = load_fb15k237()

    # Define the model and criterion
    model = TransEModel(emb_dim, kg_train.n_ent, kg_train.n_rel,
    dissimilarity_type='L2')
    optimizer = Adam(model.parameters(), lr=lr, weight_decay=1e-5)
    trainer = Trainer(model, MarginLoss(margin), kg_train, n_epochs,
                      batch_size, optimizer=optimizer,
                      sampling_type='bern', use_cuda=use_cuda,)

    trainer.run()

    evaluator = LinkPredictionEvaluator(model, kg_test)
    evaluator.evaluate(200)
    evaluator.print_results()


if __name__ == "__main__":
    if cuda.is_available():
        main(use_cuda="all")
    else:
        main(use_cuda=False)
```

Figure 4.3: Training TransE on FB15k237 using a wrapper.

```
a := [1, 2, 3, 4]
b := [5, 6, 7, 8]
c := [0, 0, 0, 0]
for i:= 0 to 3 do:
    c[i] := a[i] + b[i]
```

(a) Sequential code.

```
a := [1, 2, 3, 4]
b := [5, 6, 7, 8]
c := a + b
```

(b) Vectorized code.

Figure 4.4: Sequential and vectorized code for the addition entry-wise addition of two lists.

Continuous Delivery

Continuous Delivery (CD) on the other hand consists in frequent releases of software with a coherent versioning system: three digits for major, minor and patch release. A usual convention is to certify backward compatibility within major versions. TorchKGE has however not yet reached the first major release because some changes to the API (e.g. renaming of methods of interfaces) are still likely to occur[8]. CD also recommends the use of automated workflows to allow automatic code testing and compatibility checks upon branch merges and releases. The development workflow of TorchKGE uses GitHub actions for automation.

Eventually, TorchKGE is publicly released under a BSD license and distribution is done using Pypi, one of the main package manager in Python. This makes the library very easy to install. The PyTorch dependency should however be installed a priori to ensure CUDA compatibility and GPU support.

```
$ pip install torchkge
```

Vectorization

Vectorized code refers computation that are done on all components of a vector at once, as opposed to sequential and slower operations done by looping over the entries. A simple example of vectorization is presented in Figure 4.4. In Python, the `for` loop is not natively parallelized, which is better for simplicity of use, but it makes the sequential code of Figure 4.4a slower than the one of Figure 4.4b in which each additions, which are independent, can be done simultaneously. `Numpy`'s central `array` object support vectorized operations and of course, tensor computing frameworks like PyTorch and TensorFlow also support it with their `tensor` objects. A coding guideline of TorchKGE is to vectorize as much operations as possible and use `for` loops only if necessary.

4.4 Code structure

TorchKGE is organized in separate modules and the main ones are `models`, `evaluation`, `sampling` and `utils`. Here are the details of these modules.

4.4.1 Models

All model implementations and interfaces are in the `models` module. In order to make the most of the auto-grad function of PyTorch seamlessly on the models parameters, the core model interface `models.interfaces.Models` inherits from `torch.nn.Module` and requires the implementation of different methods.

[8] Waiting a long time before the first major release is not uncommon. For example, it took `Pandas`, one of the most common Python library, 10 years to reach that stage.

```
class Model(torch.nn.Module)
    ...

    def forward(self, heads, tails, relations, negative_heads,
                negative_tails, negative_relations=None):
        if negative_relations is None:
            negative_relations = relations
        pos = self.scoring_function(heads, tails, relations)
        neg = self.scoring_function(negative_heads,
                                    negative_tails,
                                    negative_relations)

        return pos, neg
```

Figure 4.5: `torchkge.models.interfaces.Model` implementation of the forward method required by the `torch.nn.Module` inheritance.

Scoring function

The `scoring_function` method should compute the score for each fact of a batch. Of course, its implementation should be vectorized to support GPU acceleration easily. The inheritance from `torch.nn.Module` designates the parameters that require gradient descent. The interface also requires the definitions of a `forward` method that is used to automatically compute the gradient of the loss with respect to the parameters of the model. As stated in Chapter 3, the scoring function fully defines each model's specificity and it is then used in the implementation of the `forward` method. Eventually, as loss functions used to train KG embedding models always depend on the models parameters only through the scoring function, it can be implemented once and for all in the interface (cf. Figure 4.5).

Vectorizing inference

Inference is the use a model to make a choice between possible candidates. In link-prediction for example, all entities must be ranked as possible head and tails, whereas in relation-prediction, the relations have to be ranked. A `inference_scoring_function` method is then required to compute the score of all possible candidate triples for each fact of a batch. This is a generalization of the previous scoring function method, which is needed for very fast link-prediction and relation-prediction evaluations.

For better understanding, let us define a running example, which will be used for illustration: the tail test of a link-prediction evaluation. Let n_e and n_r be the number of entities and relations in the KG, let b be the size of the current batch of triples and d the dimension of the embedding. Let $(E, R) \in \mathbb{R}^{n_e \times d} \times \mathbb{R}^{n_r \times d}$ be the tensors of all entity and relation embeddings and $(\boldsymbol{h}, \boldsymbol{t}, \boldsymbol{r}) \in (\mathbb{R}^b)^3$ be the tensors containing the indexes of the heads tails and relations of the current batch of triples.

The method `scoring_function` is able to compute the scores of the b triples of the batch in a vectorized way with simple operations on tensors of the same shape: $E[\boldsymbol{h}]$, $E[\boldsymbol{t}]$ and $R[\boldsymbol{r}]$. However, in the tail test, all entities need to be scored as possible tails for each of the b pairs of head and relation provided. That makes $b \times n_e$ scores to compute.

A naive solution would be to loop over all possible head/relation pairs and to compute the scores using `scoring_function` with repeated entries (cf. Figure 4.6). This would be inefficient because it comes down to a sequence of b vectorized computations.

It is possible to use the vectorization of `scoring_function` to fully vectorize the inference by calling it on index tensors with a lot of repeated entries, namely $\boldsymbol{h}$ and $\boldsymbol{r}$ will be repeated n_e times and $\boldsymbol{e}$ will be repeated b times. This data redundancy can generally be handled so that no useless parameter copy is made, thus avoiding memory overflow. It can become very tricky with complex models, however. Moreover, it is still inefficient

```python
def sequential_evaluation(h, r, e):
    """
    h: torch.Tensor, shape=(b, d)
    r: torch.Tensor, shape=(b, d)
    e: torch.Tensor, shape=(n_e, d)
    """
    b, d = h.shape
    scores = []
    for i in range(d):
        current_scores = []
        h_tmp = h[i].view(1, d).repeat(n_e, d)
        r_tmp = r[i].view(1, d).repeat(n_e, d)
        current_scores.append(torch.norm(h_tmp + r_tmp - e, dim=1))
    return torch.cat(scores, dim=0))
```

Figure 4.6: Sequential inference.

because many models require pre-processing of their parameters (e.g. projection in relation-specific subspace or relation matrix building) and this method would repeat the pre-processing each time an index entry is repeated. A caching method could be considered but memory management is very tricky, especially with GPU support if we want it to apply to any possible model.

The solution implemented in TorchKGE keeps things simple by defining a new object method for each model called `inference_scoring_function`. This method can compute the scoring function in inference mode by handling three dimensional tensors. Those three dimensions are usually the batch size, the embedding dimension and lastly the number of entities in the KG. Those tensors are prepared by another required method called `inference_prepare_candidates` that pre-processes embeddings only once. They are then used in higher-dimensional tensors by paying careful attention to avoid useless data recopy. To that prospect, useful tensor methods in PyTorch are `view` and `expand`. They act like reshaping and repeating methods but without copying any data.

If the evaluated model is TransH for example, the embeddings of the head entities can be projected only once on the relation-specific hyper-planes and then expanded n_e times (without additional memory usage) along the third dimension of a tensor. Eventually tensor addition and norm computations are done using built-in vectorized functions from PyTorch. A partial implementation of TransH is presented in Figure 4.8 as an example, it does not implement relation inference. Some caching is still used to further speed-up evaluation of some specific models that require a lot of pre-processing, e.g TransR.

Pre-implemented models

As of version 0.17.7, TorchKGE features the following pre-implemented models:

- 5 translational models: TransE, TransH, TransR, TransD and TorusE [17, 131, 78, 60, 41]
- 5 bilinear models: RESCAL, DistMult, HolE, ComplEx and ANALOGY [95, 141, 94, 122, 80]
- 1 deep model: ConvKB [93]

Pre-trained models

Some of the previous models were trained on usual datasets and have been made available directly from the library with functions from `torchkge.utils.pretrained_models`. New ones are also continuously added. See Table 4.2 for a list of available pre-trained models. WDV5, which will be properly introduced in Chapter 5, is a subset of Wikidata corresponding to the entities corresponding to the level 5 of Wikipedia vital articles[9].

[9] https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Level/5

	FB15k237		WN18RR		YAGO3-10	
	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
RESCAL	0.307	0.496	0.424	0.483	0.334	0.560
TransE	0.286	0.457	0.236	0.501	0.261	0.466
ComplEx	0.308	0.481	0.455	0.508	0.421	0.602

Table 4.1: Performance metrics of pretrained models

	FB15k	FB15k237	WN18	WN18RR	YAGO3-10	WDV5
RESCAL	✗	✓	✗	✓	✓	✗
TransE	✓	✓	✓	✓	✓	✓
ComplEx	✗	✓	✗	✓	✗	✓

Table 4.2: List of available datasets for TransE and ComplEx pre-trained models.

These pre-trained models were trained using usual optimization techniques such as early-stopping, dropout, weight decay and learning-rate decay. These techniques are detailed in Appendix A. Hyper-parameters we selected using the Bayesian optimization strategy proposed by `Hyperopt`[10]. They were also trained using a reasoning approach, resembling the ones presented at the end of Chapter 3. The set of training facts are enriched a priori with the triples generated by grounding rules mined by AMIE3 [73]. The confidence threshold becomes an hyper-parameter of the model chosen in $\{0.5, 0.8, 0.9, 1\}$.

Table 4.1 reports the filtered performances of some pre-trained models. It is interesting to note that, despite being the oldest model, RESCAL performs largely better than TransE on almost all the metrics and comparably to ComplEx. In the original papers, which only included experiments on FB15k and WN18, TransE [17] was better than RESCAL and ComplEx [122] was better than TransE.

4.4.2 Evaluation module

The `torchkge.evaluation` module features three evaluation techniques, implementing the methods presented in Section 3.2.4.

Triple Classification

Introduced by [111], it evaluates the quality of the embedding by measuring how well the model classifies triples as right or wrong. For initialization, the object needs a trained model, a validation and a test set. On the validation set, score thresholds are computed for each relation so that any fact with a score higher than the relation-specific threshold should be true. The thresholds are then used to classify as true or false the triples of the test set and its negatively sampled version. The metric reported is the accuracy. This process is done using three consecutive methods of the `evaluation.TripletClassificationEvaluator` object.

- `get_scores`: compute the score of all the facts using the `scoring_function` method of the model.
- `evaluate`: get the relation-specific thresholds on the validation set.
- `accuracy`: compute the accuracy of the binary classification of testing facts using the previously computed thresholds.

[10] https://hyperopt.github.io/hyperopt/

Link-prediction and relation-prediction

These methods evaluate the quality of the embedding by measuring how well the model predicts respectively missing entities and relations in facts. It is necessary to compute the scores of all candidate triples for each head, tail and relation test associated to the facts of the KG at hand. These scores are computed with the `inference_scoring_function` method of the model. Once the recovery ranks of the true heads, tails and relations are computed, the module can return the three standard metrics (MR, MRR, Hit@k) in raw and filtered settings [17].

As explained in Section 4.4.1, this evaluation process is done very efficiently thanks to the required definition of the `inference_scoring_function` method. While most frameworks loop over the facts of a KG to compute the scores of related candidate triples, this method makes it possible to group those computations by batch, dramatically increasing the computation speed. It is also further-accelerated by pre-computing the model specific projections when possible (e.g. TransR). See Section 4.5 for details on the performance of this module.

4.4.3 Knowledge graphs in memory

TorchKGE requires a representation of knowledge graphs in memory. This is the purpose of the `data_structures.KnowledgeGraph` class. It features a `split_kg` method to split the facts of the knowledge graph in train, test and optionally validation sets. When the split is random, the method keeps at least one fact involving each entity and each relation in the training subset.

The `utils.datasets` module provides functions to easily load the datasets presented in Chapter 3 among others: FB13, FB15k, WN18, FB15k237, WN18RR , YAGO3-10, WikiDataSets and WDV5 [17, 119, 35, 19, 20].

Eventually, TorchKGE also implements tools to measure data redundancy in any dataset. Those tools are extracted from the study presented in 2020 by Akrami et al. in [2] that has already been discussed in Chapter 3.

4.4.4 Negative sampling

Negative sampling is the generation of false triples by corruption of known facts and using randomness. As discussed in Chapter 3, it is instrumental in the generation of relevant embeddings. TorchKGE implements various negative sampling methods as different object classes inheriting from the interface `sampling.NegativeSampler`. This interface requires its child classes to implement the `corrupt_batch` method, which corrupts the batch of facts passed as arguments and that will be used during training. It also helps define the `corrupt_kg` method, which can be used to corrupt all the facts of a KG at once.

Currently, the `torchkge.sampling` module features the following negative sampling objects in the `sampling` module, three of which have been presented previously. The last one will be introduced in Chapter 5.

- `UniformNegativeSampler` [17]
- `BernoulliNegativeSampler` [131]
- `PositionalNegativeSampler` [111]
- `BernoulliRelationNegativeSampler` [20]

	TransE	TransD	RESCAL	ComplEx
Number of batches	10		20	10
Embedding dimension	100			
Hidden dimension		50		
Loss	Margin loss		Sigmoid loss	
Margin	1			

Table 4.3: Hyper-parameters used to train the models in all frameworks of the experiment.

4.5 Performances

4.5.1 Experimental setup

OpenKE, AmpliGraph and TorchKGE were compared in terms of running times. First pykg2vec (v0.0.50) was supposed to be included in the comparison but we did not manage to make the library work sufficiently well, due to its unstable API and outdated documentation. Four different models (TransE, TransD, RESCAL and ComplEx) were trained in each framework (apart from AmpliGraph which does not feature TransD and RESCAL) on both FB15k and WN18 and then evaluated by link-prediction on test sets. The goal was to measure the mean epoch time and evaluation duration in each case. Even if those two datasets are flawed (cf. Chapter 3), the competition here is only computational and could basically be run on automatically generated toy datasets.

Experiments were done with AmpliGraph 1.3.1, TorchKGE 0.16.0, PyTorch 1.5, TensorFlow 1.15 and a Tesla K80 GPU powered by Cuda 10.2. The version of OpenKE was the one of April 9, 2020 on GitHub. For the CPU parallelized part of OpenKE, 8 threads were used. For comparison purpose, the same sets of hyper-parameters were used in each framework: Adam optimizer, 0.01 as learning rate, 1e-5 as L2-regularization factor, Bernoulli negative sampling and one negative sample per triplet. Model specific hyper-parameters are detailed in Table 4.3.

4.5.2 Results

Table 4.4 shows for each library the mean duration of the first 100 epochs of training. They were averaged over 10 independent training procedures. The mean duration of an epoch is however quite stable: the standard deviation was always less than 0.007 seconds, which is not significant compared to the average duration. In the case of RESCAL, TorchKGE was able to train using only 10 batches thanks to lower memory usage. It could then be faster than OpenKE, which needed 20 batches. Both frameworks were eventually trained with 20 batches to be comparable. In term of model training, the implementations proposed in TorchKGE are comparable to the ones of other libraries. The mean epoch times are of the same order and always within a factor 2.

Table 4.5 reports the duration of the link-prediction evaluation process for each model in each framework. Each duration was averaged over 5 independent evaluation processes. Once again, they are quite stable: the standard deviations were always not significant compared to the average. These results show that TorchKGE is significantly faster than the two other frameworks: always at least three times faster and up to twenty-four times in the case of OpenKE when evaluating RESCAL on WN18. This shows that the initial goal of providing users with a fast evaluation module is reached.

	TransE		TransD		RESCAL		ComplEx	
	FB15k	WN18	FB15k	WN18	FB15k	WN18	FB15k	WN18
AmpliGraph	**0.171**	**0.106**					0.528	0.254
OpenKE	0.415	0.150	**0.538**	**0.284**	**3.46**	**1.05**	0.457	**0.191**
TorchKGE	0.312	0.156	0.770	0.543	3.91	1.20	**0.449**	0.287

Table 4.4: Mean epoch duration (in seconds) of 10 independent training processes. Standard deviations were all smaller than 0.007, which is not significant. Best values are in bold.

	TransE		TransD		RESCAL		ComplEx	
	FB15k	WN18	FB15k	WN18	FB15k	WN18	FB15k	WN18
AmpliGraph	354.8	39.8					537.2	94.9
OpenKE	235.6	42.2	258.5	43.7	789.1	178.4	354.7	63.9
TorchKGE	**76.1**	**13.8**	**60.8**	**11.1**	**46.9**	**7.06**	**96.4**	**18.6**

Table 4.5: Duration (in seconds) of the link-prediction evaluation process on test sets. Those figures are averaged over 5 independent evaluation processes (standard deviation was never significantly high). Best values are in bold.

4.6 Future of Developments

When it was launched, TorchKGE set the standard for fast inference in KG embedding frameworks. It should still grow to reach a critical community size and attention must be paid specifically to growing the community of contributors. Indeed, despite the rather large attention it drew, with more than 300 stars on GitHub, tens of forks and issues raised, the library received less than ten contributions from external teams through the pull request system. Successfully encouraging contributions is undoubtedly a key to further developing the adoption of TorchKGE in the research community.

Several further developments of the library are also possible. Of course, the set of pre-implemented models could be enriched, so is the set of losses and regularizers. Another possibility is to include new modules for entity typing. Many tools were already developed relatively to the work presented in Chapter 6 and are to be released soon.

Another nice-to-have possible development would be the removal of the method `inference_scoring_function` from the `model` interface. Of course, this has to be done without any speed loss in inference. The risk, however, is that it might bring a lot of complexity to an API that has been kept easy to read, understand and modify. This is indeed a differentiating strength of TorchKGE, particularly when comparing it to Pykeen. The latter grew significantly since TorchKGE was launched and it is now the most extensive alternative thanks to a large collection of models and tools. An interesting advantage compared to TorchKGE is that it also proposes a fast evaluation module but without requiring any helper scoring function, thanks to a very high-end caching system. Comparative performances are reported in Table 4.6. The performance and extensive catalog of tools however come at the cost of code simplicity. A simple API, easy to tweak and experiment with is the strength TorchKGE will have to put forward in the future to keep thriving.

	TransE		TransD		RESCAL		ComplEx	
	FB15k	WN18	FB15k	WN18	FB15k	WN18	FB15k	WN18
TorchKGE	76.1	13.8	**60.8**	**11.1**	46.9	7.06	96.4	18.6
Pykeen v1.9.0	**43**	**5**	112	21	**36**	**3**	**77**	**13**

Table 4.6: Duration (in seconds) of the link-prediction evaluation process on test sets. Those figures are averaged over 5 independent evaluation processes (standard deviation was never significant). Best values are in bold.

```python
import torch
from ignite.engine import Engine, Events
from ignite.handlers import EarlyStopping
from ignite.metrics import RunningAverage
from torch.optim import Adam

from torchkge.evaluation import LinkPredictionEvaluator
from torchkge.models import TransEModel
from torchkge.sampling import BernoulliNegativeSampler
from torchkge.utils import MarginLoss, DataLoader, load_fb15k237


def process_batch(engine, batch):
    h, t, r = batch[0], batch[1], batch[2]
    n_h, n_t = sampler.corrupt_batch(h, t, r)

    optimizer.zero_grad()

    pos, neg = model(h, t, r, n_h, n_t)
    loss = criterion(pos, neg)
    loss.backward()
    optimizer.step()

    return loss.item()


def linkprediction_evaluation(engine):
    model.normalize_parameters()

    loss = engine.state.output

    # validation MRR measure
    if engine.state.epoch % eval_epoch == 0:
        evaluator = LinkPredictionEvaluator(model, kg_val)
        evaluator.evaluate(b_size=256, verbose=False)
        val_mrr = evaluator.mrr()[1]
    else:
        val_mrr = 0

    print('Epoch {} | Train loss: {}, Validation MRR: {}'.format(
        engine.state.epoch, loss, val_mrr))

    try:
        if engine.state.best_mrr < val_mrr:
            engine.state.best_mrr = val_mrr
        return val_mrr

    except AttributeError as e:
        if engine.state.epoch == 1:
            engine.state.best_mrr = val_mrr
            return val_mrr
        else:
            raise e
```

```
device = torch.device('cuda')

eval_epoch = 20  # do link-prediction evaluation every 20 epochs
max_epochs, patience = 1000, 40
emb_dim, lr = 100, 0.0004

kg_train, kg_val, kg_test = load_fb15k237()
model = TransEModel(emb_dim, kg_train.n_ent, kg_train.n_rel, 'L2')
model.to(device)

optimizer = Adam(model.parameters(), lr=lr, weight_decay=1e-5)
criterion = MarginLoss(margin=0.5)
sampler = BernoulliNegativeSampler(kg_train, kg_val=kg_val, kg_test=
    kg_test)

trainer = Engine(process_batch)
RunningAverage(output_transform=lambda x: x).attach(trainer, 'margin')
handler = EarlyStopping(patience=patience,
                        score_function=linkprediction_evaluation,
                        trainer=trainer)
trainer.add_event_handler(Events.EPOCH_COMPLETED, handler)

# Training
train_iterator = DataLoader(kg_train, 32768, use_cuda='all')
trainer.run(train_iterator,
            epoch_length=len(train_iterator),
            max_epochs=max_epochs)

print('Best score {:.3f} at epoch {}'.format(handler.best_score,
                                             trainer.state.epoch - handler
    .patience))
```

Figure 4.7: Training a TorchKGE model using early-stopping in Ignite.

```python
from torch import empty
from torch.nn import Parameter
from torch.nn.functional import normalize

from torchkge.models.interfaces import Model
from torchkge.utils import l2_dissimilarity, init_embedding


class TransH(Model):
    def __init__(self, emb_dim, n_entities, n_relations):
        super().__init__(n_entities, n_relations)
        self.dissimilarity = l2_dissimilarity
        self.emb_dim = emb_dim
        self.ent_emb = init_embedding(self.n_ent, self.emb_dim)
        self.rel_emb = init_embedding(self.n_rel, self.emb_dim)
        self.norm_vect = init_embedding(self.n_rel, self.emb_dim)

        self.normalize_parameters()

        self.evaluated_projections = False
        self.projected_entities = Parameter(empty(size=(self.n_rel,
                                                        self.n_ent,
                                                        self.emb_dim)),
                                            requires_grad=False)

    def scoring_function(self, h_idx, t_idx, r_idx):
        self.evaluated_projections = False

        h = normalize(self.ent_emb(h_idx), p=2, dim=1)
        t = normalize(self.ent_emb(t_idx), p=2, dim=1)
        r = self.rel_emb(r_idx)
        norm_vect = normalize(self.norm_vect(r_idx), p=2, dim=1)

        return - self.dissimilarity(self.project(h, norm_vect) + r,
                                    self.project(t, norm_vect))

    def inference_prepare_candidates(self, h_idx, t_idx, r_idx):
        if not self.evaluated_projections:
            self.evaluate_projections()

        r = self.rel_emb(r_idx)
        proj_h = self.projected_entities[r_idx, h_idx]
        # shape=(b_size, emb_dim)
        proj_t = self.projected_entities[r_idx, t_idx]
        # shape=(b_size, emb_dim)
        candidates = self.projected_entities[r_idx]
        # shape=(b_size, self.n_rel, self.emb_dim)

        return proj_h, proj_t, r, candidates

    def inference_scoring_function(self, proj_h, proj_t, r):
        b_size = proj_h.shape[0]

        if len(proj_t.shape) == 3:
            assert (len(proj_h.shape) == 2)
            # this is the tail completion case in link-prediction
            hr = (proj_h + r).view(b_size, 1, r.shape[1])
            return - self.dissimilarity(hr, proj_t)
        else:
            assert (len(proj_h.shape) == 3) & (len(proj_t.shape) == 2)
            # this is the head completion case in link-prediction
            r_ = r.view(b_size, 1, r.shape[1])
            t_ = proj_t.view(b_size, 1, r.shape[1])
            return - self.dissimilarity(proj_h + r_, t_)
```

Figure 4.8: TransH partial implementation. Only head and tail inference are implemented in this example. Support for relation inference is also needed and can be found in the real TorchKGE implementation.

Chapter 5

Automatically enriching Wikidata using Wikipedia hyperlinks

This chapter presents a method to predict missing facts in Wikidata by doing relation-prediction on candidate pairs of entities. The pairs are likely to be involved in a fact as they come from hyperlinks linking Wikipedia pages.

This chapter covers the following article:

Enriching Wikidata with Semantified Wikipedia Hyperlinks.
Boschin, A., Bonald, T. (2021)
Postprint presented at the *2nd Wikidata Workshop co-located with the 20th International Semantic Web Conference* (October 2021).
Reference [20]

5.1 Motivations

Since its launch in 2001, Wikipedia[1] has successfully become the largest open-source collection of knowledge. Its textual content is however mostly unstructured, the structured information being mainly limited to the content of infoboxes (e.g. place and date of birth for articles on humans). Another structure in Wikipedia that is yet to be fully integrated in Wikidata lies in the hyperlinks between pages. The main challenge to this integration is that, to know the meaning of a hyperlink, it is necessary to read the text in which the hyperlink is embedded. While a few hyperlinks do not correspond to relevant facts, we claim that this is a rich source of information to complete Wikidata. To illustrate this, one can look at the level 5 of Wikipedia vital articles[2], that is about 40,000 pages "*serving as a centralized watchlist to track the quality of the most important articles*". These Wikipedia pages are linked by slightly more than 3 million hyperlinks, which is far more than the approximately 200,000 facts linking the corresponding entities in Wikidata. For instance, there is a link from the page *Henri Poincaré* to the page *Optics* in Wikipedia. This suggests the existence of a relation linking the two entities, here *field of work*. This fact is *not* present in Wikidata.

[1] https://www.wikipedia.org
[2] https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Level/5

Precisely, we address the task of relation-prediction: finding the relation linking some given head and tail entities. For instance, we would like to complete the triple $\langle Berlin, \_, Germany \rangle$ with the relation *capital of*, assuming the fact is not in the KG. This task, also known as the semantification of a link, was already introduced in Section 3.2.4 as an evaluation technique for KG embedding models. Though most existing works on embeddings have focused on link-prediction, we show that embeddings can also perform notably well when predicting relations. We do this by looking into a real-world application and not only the evaluation technique.

The main contributions in this chapter are the following:

- An approach to enrich Wikidata by the semantification of hyperlinks from Wikipedia. This approach relies on KG embedding models and on a symbolic type filtering.
- A novel negative sampling technique that improves the ability of KG embedding models to predict relations by balancing the role of entities and relations, without affecting their performance on link-prediction.
- A novel filtering technique for relation-prediction where candidate relations are selected through the types of the head and tail entities.
- A new dataset, WDV5, consisting of the facts between entities of Wikidata corresponding to the level 5 of Wikipedia vital articles[3]. This dataset can easily be accessed using the Python library TorchKGE.

5.2 Related work

5.2.1 Embedding and negative sampling

KG embedding was extensively introduced in Chapter 3. Let us simply recall that there are mainly three categories of models depending on the form of their scoring function f which reflects how entities and relations are modeled to interact in the embedding vector space: linear models (relations are modeled as translations from the head to the tail entity), bilinear models (relations are models as bilinear forms *distorting* the vector space in which head and tail entities interact) and deep models using neural networks to define all sorts of parametric scoring functions.

Deep models possibly include techniques from the deep learning literature such as attention mechanisms [91, 130] or rely on Large Language Models (LLMs): GilBERT [90] for example, is a more recent model that specifically tackles the task of relation-prediction by fine-tuning the BERT LLM [36]. In accordance with the conclusion of Chapter 3, the method proposed here relies on simpler models (TransE and ComplEx [17, 121]).

As explained in Section 3.2.2, false triples are required to train embedding models as their scoring function f are expected to discriminate true from false. The random generation of false triples is called negative sampling and it can have a major impact on the performance of the trained model [70]. Given some known fact $\langle h, r, t \rangle$, the usual way to create a false triple from it (under the CWA) is to randomly choose either the head entity or the tail entity and to replace it with another random entity of the KG [17]. This technique was improved in [131] by using a Bernoulli parameter. The replacement of the relation is rarely considered. It is mentioned in [137] but not precisely described and its impact not precisely measured, as it is not the focus of that article. We propose a modification of the Bernoulli NS technique to include random replacement of the relation, to get high performance in both link-prediction and relation-prediction.

[3] https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Level/5

5.2.2 Type filtering

As presented in Chapter 2, most KBs assign types to entities through a `rdf:type` relation (e.g. the *P31: instance of* relation of Wikidata). They usually come with taxonomic axioms such as `rdfs:domain` and `rdfs:range` constraints on the relations. As stated in Section 3.4.3, these taxonomic axioms have mainly been used to train models, either during negative sampling or directly by constraining the loss function. They can however also be used during inference as a criteria to select the candidate entities [71, 137].

In relation-prediction, selecting candidate relations with these axiomatic constraints seems natural but they can be missing or too coarse grained making the filtering either too restrictive or with no effect. In Wikidata, relation constraints are hints for the editors, not firm restrictions[4]. That leads to domains and ranges that are often either undocumented or heterogeneous. We propose a simple method to *infer* such constraints from the `rdf:type` relation of the KG at hand and use the resulting rules for relation-prediction. We show that it has a major impact on performance.

5.2.3 NLP for relation-prediction

When it comes to KG completion, there are two main tasks tackled by NLP methods. The first task is relation linking, that is linking relations of a KG to plain text surface forms. Some interesting articles are [106, 142]. The second a more common task is relation extraction, consisting in predicting the semantic relation linking two entities based on sentences involving these entities. The best performing models rely on deep neural network architectures with attention mechanisms [129, 102, 146] or directly on pre-trained LLMs such a LUKE [140]. The task of relation-prediction in KG, tackled in the current chapter, is different from the two previous ones as it relies only on the graph structure of the KG and not on any textual content. A method combining both graph and textual inputs is left as an interesting perspective for future research.

5.2.4 Wikipedia hyperlink semantification

Very few works exist specifically on the semantification of Wikipedia hyperlinks using only the graph structure of the KG. The closest approach was presented by Galarraga et al. in 2015 [47]. It is based on rule mining but limit of this method is that it can only predict relations for entities matching the body of the mined rule. The proposed technique based on KG embedding applies to all links.

5.3 Proposed approach

The proposed approach relies on the combined use of several techniques, which will be presented now.

5.3.1 Ranking relations

KG embedding models can be used to rank relations: given two entities h and t, an embedding model and its scoring function f, the relations of the graph can be ranked by decreasing order of scores: $f(h, r_1, t) > f(h, r_2, t) > \cdots > f(h, r_k, t)$. The relation r_1 is then predicted, corresponding to the fact (h, r_1, t). Note that this method applies to the case of undirected links, by ranking the scores of the predictions for both directed links (h, t) and (t, h). This is especially useful when some relations have no reciprocal. In practice, TorchKGE proposes an efficient implementation of ranking.

[4] https://www.wikidata.org/wiki/Help:Property_constraints_portal

5.3.2 Balanced negative sampling

A simple experiment shows that off-the-shelf linear models like TransE perform really badly in relation-prediction (3% of Hit@1 on FB15k237, cf. Table 5.3a). This suggests that the representation of relations is not as good as that of entities. It turns out that entities and relations play similar roles in the training procedure except for the NS step. In the usual NS techniques presented in Chapter 3, only the entities are randomly replaced to get false triples. We propose a simple modification of BerNS to balance the roles of entities and relations during training. Rather than just replacing one of the two entities of a known fact, the relation is replaced with some probability p, and otherwise BerNS is applied (cf. Algorithm 4). This new method is called Balanced Negative Sampling (BalNS). The default value for p is set to $\frac{1}{2}$. Experiments have shown that the value of the parameter has no major impact on the performances of the approach as long as it is in the range $[0.1, 0.7]$.

Algorithm 4: Balanced Negative Sampling (BalNS).

Input: (h, r, t), a fact
Input: p, probability to replace the relation
Output: (h', r', t'), a false triple
Data: $\mathcal{T}$, the facts in the KG
Data: p^r, Bernoulli parameter for relation r

$(h', r', t') \leftarrow (h, r, t)$
while $(h', r', t') \in \mathcal{T}$ **do**
  $u \leftarrow$ uniform random variable on $[0, 1]$
  if $u < p$ **then**
    $r' \leftarrow$ random relation
  else
    $(h', t') \leftarrow \text{BerNS}(h, r, t)$
return (h', r', t')

5.3.3 Type filtering for relation-prediction

Another key technique to improve the quality of relation-prediction is through Type Filtering (TF). An entity e is said to have the type t if the fact $\langle e, \texttt{rdf:type}, t \rangle$ is known. To predict the relation linking h and t, only relations that are known to link entities of the type(s) of h to entities of the type(s) of t should be considered. Formally, we say that a relation r links type a to type b if there exists some known fact $\langle h, r, t \rangle$ with the head entity h of type a and the tail entity t of type b. To predict the relation missing in $\langle h, \_, t \rangle$, we propose to consider as candidates only the relations r linking any type of h to any type of t. The corresponding algorithm for relation-prediction is described in Algorithm 5. Observe that if either the head entity h and/or the tail entity t is not typed, the candidate relations are then the relations that are involved in a training fact with either h as a head entity or t as a tail entity. In the end, if no relation meets any constraint, there is no filtering, i.e. all relations are selected. Regarding speed, this step has no significant impact on the global computation time if a proper index is built a priori: link between entities and their types and types to possible relations.

To summarize, our approach relies on the following steps:

1. Training the model (e.g. TransE or ComplEx) with BalNS (Algorithm 4).
2. Predicting relations with TF (Algorithm 5).

Algorithm 5: Relation-prediction with Type Filtering (TF).

Input: h, t, entities
Input: f, scoring function
Output: r, relation linking h to t
Data: $\mathcal{T}$, the facts in the KG
Data: $\mathcal{R}$, the relations in the KG

$A \leftarrow$ types of h
$B \leftarrow$ types of t
if $|A| > 0$ *and* $|B| > 0$ **then**
  $R \leftarrow \{r : \forall (a, b) \in A \times B, \exists h', t' : \text{type}(h') = a, \text{type}(t') = b, (h', r, t') \in \mathcal{T}\}$
else
  $R \leftarrow \{r \in \mathcal{R} : \exists e : (h, r, e) \in \mathcal{T}\} \cup \{r \in \mathcal{R} : \exists e : (e, r, t) \in \mathcal{T}\}$
if $|R| = 0$ **then**
  $R \leftarrow \mathcal{R}$
$r \leftarrow \arg\max(\{f(h, r, t), r \in R\})$
return r

5.4 Experimental setup

The experiments aim at assessing the performance of the proposed approach on existing KGs in a supervised way through relation-prediction (cf. Section 3.2.4) but also at showing its practical interest on the real-world task of Wikipedia hyperlinks semantification. In relation-prediction, the candidates selected by TF are ranked by decreasing score and the rank of the true relation r is recorded as the recovery rank. If the true relation r is not selected by TF, the rank is set to the maximum. All experiments can be reproduced using the publicly available code[5] and data[6].

5.4.1 Datasets

As stated in Section 3.2.5, one of the most common datasets used to evaluate the quality of KG embeddings is a subset of Freebase called FB15k237 [119]. The typing relation from Freebase is however not included in it and resources are no longer available online since the discontinuation of the Freebase project [14] in 2016. Subsequently, entity types were imported from Wikidata using a matching between the two KBs. Attention was paid to prevent data leakage by removing any imported fact that could match an existing validation or test fact. For comparability reasons, the new facts were not used to train the embedding models but only for the TF step. Only 18,6% of entities are typed, cf. Table 5.1.

We also introduce WDV5, a new dataset containing the facts linking entities of Wikidata corresponding to the level 5 of Wikipedia vital articles (cf. Section 5.1). This dataset is handful in the current chapter has it can provide performance metrics measuring if the facts predicted using Wikipedia hyperlinks are true when there is a known ground truth in Wikidata (and WDV5). We hope however that it will be widely adopted by the community to diversify the datasets used. It is important to note that WDV5 is a raw extract from Wikidata, without any pre-processing. As such, we expect the corresponding experiments to be more representative of real-world use-cases than those based FB15k237. To type entities, only typing facts included in the dataset are used (i.e. all types are entities of WDV5). In particular, not all entities are typed (only 56%, cf. Table 5.1).

For the semantification of Wikipedia hyperlinks, Wikivitals+[6] is used. This is an extraction of the level 5 of Wikipedia vital articles and the hyperlinks between them.

[5] https://gitlab.telecom-paris.fr/aboschin/hyperlinks-semantification
[6] https://netset.telecom-paris.fr/pages/wikivitals+.html

Dataset	Entities/nodes	Facts/edges	Relations	Types	Typed entities
FB15k237	14,541	310,116	237	73	2,719
WDV5	39,062	231,744	607	1,206	22,883
Wikivitals+	39,062	3,008,116			

Table 5.1: Descriptive figures of the datasets used in experiments. For FB15k237, the split between train, validation and test is fixed by literature. For WDV5, the splits are chosen randomly.

Only pages that have a corresponding Wikidata entity were kept. This dataset provides many hyperlinks that are natural candidates for true facts.

Detailed figures about the three datasets used in the experiments are available in Table 5.1.

5.4.2 Baseline

To measure the impact of using a KG embedding model for ranking the candidates selected by TF, we compare our approach to a simple baseline that ranks the candidate relations by popularity in the training set. For example, if most training facts linking a city to a country involve the relation *located in* then the baseline predicts *located in* when presented with the incomplete fact $\langle Paris, \_, France \rangle$.

5.4.3 Embedding models

Two off-the-shelf embedding models were chosen for the experiments:

- TransE [17], the simplest linear model, intuitive and fast to train and apply.
- ComplEx [120], the best bilinear model, with twice more parameters, longer to train and apply.

As explained at the beginning of this chapter, more elaborate models were excluded in accordance with the conclusion of Chapter 3.

The models were trained using the Adam algorithm for optimization, dropout and early-stopping with 100 epochs of patience (on the filtered validation MRR for link-prediction) for regularization. Details on these techniques can be found in Appendix A. All experiments were done using `Python` 3.8, `PyTorch` 1.7.0 [99], `TorchKGE` 0.16.25 [18] `pytorch-ignite` 0.4.4 and a Nvidia Titan V GPU powered with `Cuda` 10.1. The hyper-parameters of the embedding models were tuned using `hyperopt` 0.2.5. The possible values along with those chosen are listed in Table 5.2.

In the case of FB15k237, the split between train, validation and test sets is defined by Toutanova et al. [119]. For WDV5, we split the dataset at random with 80% of the facts for training, 10% for validation (for choosing hyper-parameters) and 10% for testing. The reported metrics are averaged over 6 distinct random splits and independent training procedures.

Hyper-parameter	Possible values
Learning rate	{0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05}
Batch size	{4096, 8192, 16384, 32768, 65536}
Loss type	logistic, margin, binary cross entropy
Margin (if margin loss)	{0.5, 1, 2, 3, 10}
Embedding dimension	{100, 150, 200}
Dropout	{0, 0.1, 0.2, 0.3, 0.4, 0.5}
L2 penalization	{1e-5, 5e-5, 1e-4}
LR decay	{1, 0.995, 0.99, 0.985}

(a) Possible values for the hyper-parameters.

	TransE			
	FB15k237		WDV5	
	BerNS	BalNS	BerNS	BalNS
Learning rate	0.005	0.1	NC	0.001
Batch size	4096	16384	NC	NC
Loss type	margin	margin	margin	NC
Margin (if margin loss)	1	0.5	1	NC
Embedding dimension	150	150	NC	200
Dropout	0.2	0.1	0.4	0
L2 penalization	1e-5	5e-5	NC	NC
LR decay	0.995	0.99	0.995	NC

(b) Chosen values for the hyper-parameters of TransE.

	ComplEx			
	FB15k237		WDV5	
	BerNS	BalNS	BerNS	BalNS
Learning rate	0.0005	0.0005	0.001	NC
Batch size	8192	32768	NC	NC
Loss type	margin	margin	logistic	logistic
Margin (if margin loss)	1	0.5		
Embedding dimension	200	200	NC	NC
Dropout	0.4	0.2	0.3	0.3
L2 penalization	5e-5	5e-5	1e-4	1e-4
LR decay	1.0	1.0	1.0	1.0

(c) Chosen values for the hyper-parameters of ComplEx.

Table 5.2: Possible and chosen values for the hyper-parameters. For WDV5, some values were different from one random split of the dataset to the other; NC means that the chosen values were Not Constant.

Base model	Variant	MRR	Hit@1	Hit@5
TransE	Original	0.061	0.033	0.049
	BalNS	0.940	0.914	0.972
	TF	0.405	0.184	0.744
	BalNS & TF	0.957	0.935	**0.983**
ComplEx	Original	0.928	0.894	0.967
	BalNS	0.956	0.934	0.982
	TF	0.953	0.927	**0.983**
	BalNS & TF	**0.961**	**0.943**	**0.983**
Baseline		0.153	0.050	0.262

(a) FB15k237

Base model	Variant	MRR	Hit@1	Hit@5
TransE	Original	0.556 ± 0.116	0.458 ± 0.128	0.664 ± 0.102
	BalNS	0.779 ± 0.006	0.697 ± 0.017	0.881 ± 0.012
	TF	0.711 ± 0.063	0.588 ± 0.092	0.872 ± 0.020
	BalNS & TF	0.821 ± 0.002	0.754 ± 0.003	0.903 ± 0.002
ComplEx	Original	0.546 ± 0.078	0.454 ± 0.108	0.649 ± 0.052
	BalNS	0.816 ± 0.037	0.734 ± 0.059	**0.917 ± 0.011**
	TF	0.826 ± 0.009	**0.765 ± 0.013**	0.902 ± 0.004
	BalNS & TF	**0.827 ± 0.024**	0.762 ± 0.041	0.910 ± 0.003
Baseline		0.516 ± 0.006	0.416 ± 0.006	0.618 ± 0.006

(b) WDV5 (mean ± standard deviation).

Table 5.3: Results of relation-prediction on FB15k237 and WDV5.

5.5 Results on supervised relation-prediction

The results for relation-prediction are shown in Table 5.3 for different variants of the model so as to assess the respective gains of each proposed technique. Metrics are reported in the filtered setting. The model variants are labelled as follows:

- Original: The base model (either TransE or ComplEx) trained with BerNS.
- BalNS: The base model trained with BalNS.
- TF: The base model evaluated with Type Filtering (TF).
- BalNS & TF: The base model trained with BalNS and evaluated with TF.

The figures show that the off-the-shelf version of TransE is not efficient on FB15k237 (only 3% of Hit@1). ComplEx performs however notably well on the same dataset (89% of Hit@1). It seems less sensitive to the unbalanced role of entities and relations during training. We suspect however that the score of ComplEx on FB15k237 mainly results from the over-fitting due to the issues raised in Section 3.3.4 and notably the over-engineering of FB15k237. It is confirmed by the fact that TransE and ComplEx have almost the same scores (around 45% of Hit@1) on the new dataset WDV5 which is a raw extraction from Wikidata.

Let us now investigate the impact of each technique separately.

5.5.1 Impact of balanced negative sampling

Training with BalNS has a strong impact on the relation-prediction performance of the models: training TransE on FB15k237 with BalNS rather than BerNS increases the Hit@1 from 3% to 91%. This confirms the intuition that the relation embeddings were not well trained. The difference is less impressive for ComplEx on FB15k237 but the

		FB15k237		WDV5	
Model	NS method	MRR	Hit@10	MRR	Hit@10
TransE	BerNS	0.282	0.451	0.305	0.480
	BalNS	0.286	0.457	0.296	0.465
ComplEx	BerNS	0.308	0.481	0.367	0.506
	BalNS	0.304	0.475	0.364	0.495

Table 5.4: Impact of the Negative Sampling method on link-prediction.

original ComplEx model performs already quite well on this dataset. On WDV5, there is a big increase in Hit@1 for both models: 24% for TransE and 28% for ComplEx.

Observe that this novel NS technique brings significant performance gains on relation-prediction without loss on link-prediction, as shown in Table 5.4.

5.5.2 Impact of type filtering

TF has a strong impact on the performance of the models. Looking at Hit@1 on WDV5, TransE goes from 45% to 58% and ComplEx goes from 45% to 76%. Note that Type Filtering alone (the baseline) performs almost as well as the original embedding models. It is however largely beaten by the combination of TF with scoring by an embedding model. The gain of using and embedding model is very important.

5.5.3 Results of the complete approach

The combination of BalNS and TF gives the best results in term of MRR on both datasets and with both models.

On FB15k237, the increase in performance of TransE is impressive (Hit@1 from 3% to 94%) and makes this model almost as efficient as ComplEx. This is obtained through additional facts imported from Wikidata for TF but the scores of TransE simply trained with BalNS (and without TF) are already close to those of ComplEx.

On WDV5, all performance metrics are significantly improved by our approach. Both models, that perform similarly in their original forms remain close. On average, ComplEx beats TransE by 1% in Hit@1 but the scores of TransE are much more stable from one run to the other, as shown by the lower standard deviation. The intervals of fluctuation of MRR and Hit@1 tend to be reduced if the model is trained with BalNS. This is particularly true for TransE, whose standard deviation for each metric is very small. Let us eventually notice the surprising and very slight decrease in Hit@1 (resp. Hit@5) between the TF (resp. BalNS) version of ComplEx and the complete approach. There is no clear explanation for that but the results of MRR are reassuring.

It is remarkable to get almost identical performance with TransE and ComplEx, knowing that TransE has half the number of parameters of ComplEx, is more geometrically intuitive and requires six times less operations for each gradient descent step during training. This is a confirmation of what was already stated in the conclusion of Chapter 3: simple models can perform as well as more complex ones if properly trained and combined with symbolic exploitation of the ontology.

5.6 Application to Wikipedia hyperlinks

To predict the relation associated to a Wikipedia hyperlink, we use the TransE embedding of WDV5 trained with BalNS and applied using TF. When two pages are linked and the corresponding Wikidata entities are involved in a fact of Wikidata (110,311 out of 3,008,116 hyperlinks), we can compare the predicted relation to the ground-truth. We obtain 84% of accuracy. This good score is expected as the model is trained on WDV5

facts and some hyperlinks indeed correspond to existing facts. However, it is interesting to look at cases where the prediction is different from the true fact. We have observed that the model can hardly predict directed relations (e.g. parent-child) or semantically close relations (e.g. *employer* and *educated at* for links between scholars and universities). This is not surprising as the only available data is the structure of the KG. Some other mistakes come from the embedding model itself, for example *headquarter location* always has a lower score than *twinned administrative body* for some reason, making the headquarter predictions all wrong.

In Table 5.5, we report for two pages the semantified hyperlinks that got the highest scores. It is reassuring to see that most of the resulting facts are true, many of them being however already known in Wikidata. A few mistakes could be avoided using a little bit of context (e.g. text information) but these results suggest that our approach is able to correctly semantify many links.

It seems however difficult to automatically add these predictions to Wikidata. First, the scores of embedding models are usually not normalized so comparing them works fine when done *locally* (e.g. looking at the links of a particular page) but comparing the scores of the three million possible facts is not feasible. Second, many facts that get a high score are very likely but require additional information not present in data. For example the three most likely facts resulting from semantified hyperlinks of Wikivitals+ are:

- (*Serbia*, *member of*, *World Trade Organization*): Serbia's application is still under review.
- (*Taiwan*, *member of*, *World Trade Organization*): Taiwan is already a member of the WTO through the Chinese Taipei but not in its name.
- (*Kosovo*, *member of*, *Interpol*): Kosovo's application was rejected in 2018.

Some additional textual content would be very helpful in these cases.

5.7 Conclusion

We have proposed a novel approach to improve relation-prediction by KG embedding. It is based on two key ideas: Balanced Negative Sampling in the training of the embedding model, and Type Filtering to select candidate relations during inference. We have shown that this approach performs well using an embedding model as simple as TransE. Our results suggest that the model can be used to enrich Wikidata, by the semantification of Wikipedia hyperlinks associated with known entities.

This approach is however not yet fully automatable and performances still need to be increased for that goal. A possible extension of this work is a further improvement of the negative sampling technique by replacing the relation with some probability that depends on the considered fact $\langle h, r, t \rangle$, inspired by the Bernoulli probabilities for entities. It seems also necessary to integrate some context from textual data for example (like the description of the relations and the articles themselves) to help the embedding model in its choices. A fully automatized process of enriching Wikidata with semantified Wikipedia hyperlinks seems however possible.

A possible improvement to the method is to ease the decision task by dealing with some relations in a specific way. For example, the Wikidata relation *different from* might be excluded from the training set. Another possibility is to replace it by a broader artificial relation *no-relation* that could also be involved in negatively sampled facts involving unrelated entities, or a least entities that are not known to be related. This would be closer to the functioning of the NLP task of relation extraction presented in Section 5.2.3.

Eventually, another possible use of the proposed method is to apply it to other sources

Head	Predicted Relation	Tail	Score	Evaluation
Allergy	health specialty	Immunology	-0.728593	
Allergy	has effect	Rhinorrhea	-0.839387	
Allergy	has cause	Allergen	-0.844231	
Allergy	health specialty	Pediatrics	-0.972245	
Allergy	health specialty	Internal medicine	-1.022068	
Allergy	instance of	Disease	-1.022585	
Allergy	drug used for treatment	Adrenaline	-1.040919	
Allergy	medical examinations	Blood test	-1.165760	
Allergy	drug used for treatment	Aspirin	-1.172504	
Allergy	health specialty	Hematology	-1.219639	
Allergy	possible treatment	Medication	-1.221141	
Allergy	symptoms	Abdominal pain	-1.228720	
Allergy	drug used for treatment	Penicillin	-1.235292	
Allergy	subclass of	Pollution	-1.283537	
Allergy	health specialty	Statistics	-1.297987	
Allergy	health specialty	Epidemiology	-1.301585	
Allergy	afflicts	Immune system	-1.374023	
Allergy	afflicts	Blood	-1.376165	
Allergy	possible treatment	Antibiotic	-1.406908	
Allergy	symptoms	Itch	-1.416243	

(a) Top-20 triples predicted from the page *Allergy*.

Head	Predicted Relation	Tail	Score	Evaluation
Henri Poincaré	employer	University of Paris	-0.358521	
Henri Poincaré	employer	École Polytechnique	-0.412040	
Henri Poincaré	occupation	Mathematician	-0.414613	
Henri Poincaré	place of death	Paris	-0.492610	
Henri Poincaré	occupation	Engineer	-0.537747	
Henri Poincaré	field of work	Number theory	-0.561962	
Henri Poincaré	student of	Charles Hermite	-0.592644	
Henri Poincaré	field of work	Epistemology	-0.605310	
Henri Poincaré	field of work	Topology	-0.678377	
Henri Poincaré	field of work	Algebraic geometry	-0.712538	
Henri Poincaré	student of	Wilhelm Wundt	-0.738560	
Henri Poincaré	field of work	Optics	-0.738636	
Henri Poincaré	notable work	Poincaré conjecture	-0.739285	
Henri Poincaré	field of work	Philosophy of science	-0.791077	
Henri Poincaré	field of work	Metaphysics	-0.822683	
Henri Poincaré	place of birth	Nancy, France	-0.824190	
Henri Poincaré	different from	Raymond Poincaré	-0.830676	
Henri Poincaré	student of	Karl Weierstrass	-0.843734	
Henri Poincaré	field of work	Set theory	-0.855588	
Henri Poincaré	field of work	Celestial mechanics	-0.877295	

(b) Top-20 triples predicted from the page *Henri Poincaré*.

Head	Predicted Relation	Tail	Score	Evaluation
Paris	capital of	France	-0.507053	
Paris	located in or next to body of water	Seine	-0.656630	
Paris	instance of	Capital city	-0.699302	
Paris	twinned administrative body	Rome	-0.708570	
Paris	official language	French language	-0.709513	
Paris	twinned administrative body	Berlin	-0.761601	
Paris	twinned administrative body	Strasbourg	-0.791677	
Paris	history of topic	History of Paris	-0.795585	
Paris	head of government	Jacques Chirac	-0.813331	
Paris	twinned administrative body	Madrid	-0.824193	
Paris	twinned administrative body	Cologne	-0.836061	
Paris	instance of	Administrative division	-0.839149	
Paris	continent	Europe	-0.844736	
Paris	twinned administrative body	Milan	-0.855923	
Paris	twinned administrative body	Geneva	-0.857794	
Paris	twinned administrative body	Florence	-0.864486	
Paris	shares border with	Marseille	-0.870625	
Paris	shares border with	Tel Aviv	-0.878915	
Paris	significant event	Storming of the Bastille	-0.883031	
Paris	twinned administrative body	London	-0.899031	

(c) Top-10 triples predicted from the page *Paris*. As a city can only be the capital of one country, only the prediction of the form $\langle Paris, capitalOf, \_ \rangle$ with the highest score was kept.

Table 5.5: Relation-prediction applied to the semantification of Wikipedia hyperlinks (green = true fact unknown by Wikidata, blue = fact already known by Wikidata, red = false triple).

of untyped links. In the spirit of relation extraction presented in Section 5.2.3, textual data can also be used to provide candidate entities that are likely to be in relation. A simple approach would apply the proposed technique to predict relations linking entities occurring in common sentences, relying on a pre-processing step of named-entity recognition. This would be comparable to the recent technique of [90], which relies on a BERT LLM [36] to detect if a relation exists between two entities of a given sentence and then uses KG concepts of nearest neighbors to choose among a set of possible relations.

Chapter 6

Hierarchical classification for entity typing

This chapter focuses on hierarchical classification and specifically on the possible impact a label hierarchy can have on classification. This is motivated by entity typing in KGs, which is another way of automatically completing KBs. The focus will however be on the general ML task.

The structure of the chapter is the following. A first section introduces hierarchical classification by linking it to entity-typing. Then, three methods are considered as ways to include the hierarchy in the training process of classification methods. The first one extends the existing Dirichlet graph classifier using an intuitive local-classifier-per-parent node technique, which will be presented. Then an hierarchical innovative training loss for neural and tensor-based models is introduced and experimented with. Eventually, the contradictory intuition that, conditionally to rich data, a *flat* classifier can learn the hierarchy directly from the features is reinforced by the results of a new Self-Encoder model we propose.

The Self-Encoder model is the subject of the following preprint article:

A Self-Encoder for Learning Nearest Neighbors.
Boschin, A., Bonald, T., Jeanmougin, M. (2023)
Under review at ECML23
Reference [21]

6.1 Hierarchical classification

The current section first recalls the definition of a taxonomy and then reviews some important contributions on automatic entity typing. The rest of the section introduces the underlying and more general task of Hierarchical Classification (HC), which is the focus of the chapter.

6.1.1 Taxonomies

As explained in Chapter 2, entities in KBs are typed and ontologies define a subsumption binary relation between types that yields a natural hierarchy: the taxonomy. This binary relation on classes is denoted $\subset$: $c_i \subset c_j$ if class i is a sub-class of class j. The taxonomy is a directed acyclic graph and in the simpler case where an entity can only have a single super-type, it is a tree. Figure 6.1 shows an extract of the YAGO4 taxonomy [101].

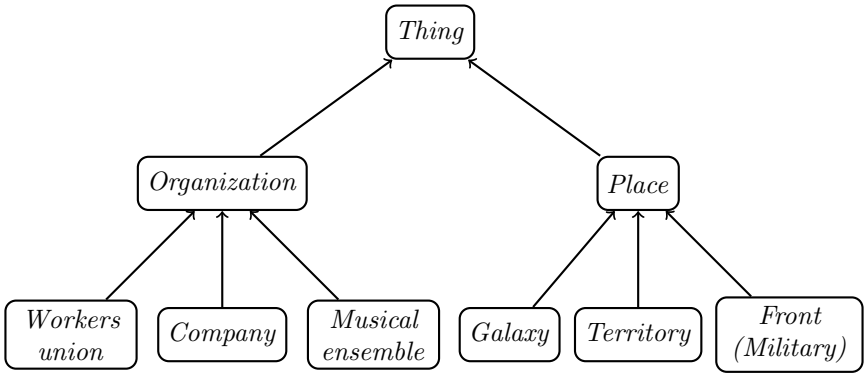


Figure 6.1: Extract of the YAGO4 taxonomy. Only the labels of the types are shown for readability.

Entity typing is the task of assigning a class of the taxonomy to existing entities in the KG. The definition of taxonomy in the context of KBs (cf. Definition 6) is actually general to classification. Entity typing is then a sub-problem of HC.

Automatic entity typing in KGs has been an active research topic for a few years and several machine learning solutions have been proposed. For example, [138] proposes FIGMENT, a method relying on word embeddings from contextual descriptions of entities. Other methods combine several data representations (e.g. graph and KG embeddings, word embeddings and other node features) to reduce the amount of textual data required and to improve the performances. Examples of such methods are MuLR [139], APE [61], Cat2Type [11] and GRAND [10]. More complex neural network structures, such as attention mechanisms and graph convolutions, have also been experimented with [62, 133, 149]. Eventually let us mention an interesting method involving restricted Boltzmann machines [132].

In the rest of the chapter, types are indifferently called *types* or *classes* and the taxonomy is supposed to be a tree, with one root class (a class with no super-class) and several leaves (classes with no sub-class).

6.1.2 Global and local hierarchical classification models

As introduced in Chapter 3, a classification model is an algorithm that is trained to assign a class to unknown samples. If there are more than two possible classes, the task is called multi-class and when the answer can be made of several classes for each sample, it is called multi-label.

The set of possible classes either comes flat or with a taxonomy. Let us call *flat* a model that has been trained with no structural information on the classes, as opposed to a *hierarchical* one. It can be discussed that HC is multi-label because if a sample is assigned a class, then it is also implicitly assigned all its super-classes up to the root. Multi-label hierarchical classification, however, usually designates the case where several unrelated classes can be assigned to a single sample. In the rest of the chapter, only *single-label* multi-class classification (hierarchical or flat) will be discussed.

Following [110], there is a natural division of HC models between the *local* and the *global* ones.

- A local approach is made of several flat classifiers. The Local Classifier per Node (LCN) mode counts as many binary classifiers as there are classes and each one decides to assign or not a specific class to samples. Another mode is called Local Classifier per Parent Node (LCPN) and associates one classifier per non-leaf class, each one deciding which of the sub-classes should be assigned to samples.

There are two ways to run local methods. First, all the classifiers can be run independently and the possible hierarchy conflict can be resolved at the end. Another more intuitive way is to apply classifiers in a sequence following a graph traversal of the taxonomy, starting from the root node and following the predictions made on the go. An example of LCPN will be described in Section 6.2.

- A global approach is made of a single classifier that somehow takes the hierarchy into account during training or when choosing among all the classes available. An example of global approach will be described in Section 6.3.

In HC, performance metrics should reflect how predictions comply with the hierarchy. They will be detailed in the following section.

6.1.3 Performance metrics

Let $C = \{c_1, \dots, c_K\}$ be a set of classes and let $(\boldsymbol{x}_i, y_i)_{i=1}^n$ be the data at hand such that $\forall i \in \{1, \dots, n\}$, $y_i \in C$. For all $i \in \{1, \dots, n\}$, the prediction associated to $\boldsymbol{x}_i$ is noted $\hat{y}_i$.

Accuracy

A key metric in classification is the accuracy, that is the share of labels correctly predicted:

$$Acc = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{y_i = \hat{y}_i\}$$

Precision, recall and f_1-score

Three usual metrics are precision, recall and f_1-score. Let us define them in the binary case followed by the multi-class case.

- In binary classification, $C = \{0, 1\}$. The predictions can intuitively be organized in four categories, as in the confusion matrix in Table 6.1: True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). The precision, recall and f_1-scores are then defined as follows:
  - Precision is the share of samples predicted as positive that are correctly classified
  $$P = \frac{|TP|}{|TP| + |FP|}$$
  - Recall is the share of samples truly positive that are correctly classified
  $$R = \frac{|TP|}{|TP| + |FN|}$$
  - f_1-score is a common aggregated measure defined as the harmonic mean of precision and recall
  $$f_1 = 2 \cdot \frac{P \cdot R}{P + R} = \frac{2 \cdot TP}{n + TP - TN}$$
- In multi-class classification ($|C| > 2$), the confusion matrix is larger but it is built in the same way. There are two possible definitions of the metrics:
  - In the micro setting (noted μ), the number of TP, FN and FP are summed over all classes. The number of TP is simply the number of correctly classified samples. The numbers of FN and FP are both equal to the number of incorrectly classified samples. Indeed, if $y_i = c_1$ and $\hat{y}_i = c_2$ then this is a FP for the class c_2 and a FN for the class c_1. As a result, precision and recall

	$y=1$	$y=0$
$\hat{y}=1$	TP	FP
$\hat{y}=0$	FN	TN

Table 6.1: Confusion matrix of a binary classification problem.

are equal in micro setting and $f_1^\mu = P = R$. Eventually, there are no TN in multi-class because each sample has exactly one class. As a result, f_1^μ is also equal to the accuracy.

– In the macro setting (noted M), each metric is computed separately for each class in a *one-vs-all* binary setup (one label against all the others) and are then averaged. Specifically, the macro f_1-score (noted f_1^M) is the average of all the binary f_1-scores:

$$f_1^M = \frac{1}{|C|} \sum_{c \in C} f_1^{(c)}$$

It is also possible to weight this average, by the support of each class for example.

Hierarchical versions

Assume now that C comes with a taxonomy. According to Kosmopoulos et al. [69], there are three main performance metrics in multi-class HC and they are *set-based*, meaning that they rely on the definition of sets of predictions for each sample. Precisely, for each sample i, the set-based metrics define y_i^{aug} (resp. $\hat{y}_i^{aug}$) as a set containing y_i (resp. $\hat{y}_i$) and other related classes from the taxonomy. The taxonomic sub-classes or super-classes can be added to these augmented sets. Sometimes, the super-classes up to a common ancestor between the true and predicted classes are added. Eventually, hierarchical precision, recall and f_1-score are computed for each sample as follows and the global metrics are simply computed by averaging those over all the samples.

$$P^h(\hat{y}_i^{aug}, y_i^{aug}) = \frac{|\hat{y}_i^{aug} \wedge y_i^{aug}|}{|\hat{y}_i^{aug}|}$$
$$R^h(\hat{y}_i^{aug}, y_i^{aug}) = \frac{|\hat{y}_i^{aug} \wedge y_i^{aug}|}{|y_i^{aug}|}$$
$$f_1^h(\hat{y}_i^{aug}, y_i^{aug}) = 2 \cdot \frac{P_h \cdot R_h}{P_h + R_h}$$

The rationale behind the augmented sets is that predictions should not be considered as standalone but within the taxonomy. Using the extract taxonomy in Figure 6.1, if a model predicts the class *Company* for a sample while the true class was the super-class *Organization*, then it should be less penalized than if the prediction was *Place* and more penalized than if the prediction was *Thing*.

6.2 Dirichlet node classification

This section introduces the Dirichlet classifier following [33] and a hierarchical version using the LCPN method.

An interesting feature of the Dirichlet classifier is that it does transductive [50] learning in the sens that its learning process uses the entire adjacency matrix of the underlying graph and not simply the feature vectors (e.g. the rows of the adjacency matrix) of the labeled nodes of the training set. That usually makes it an excellent choice to solve graph learning tasks, like node classification.

6.2.1 Original model

The Dirichlet classifier relies on the physical mechanism of heat diffusion, which is ruled by the heat equation $\frac{\partial T}{\partial t} = \alpha \Delta T$ where T is the temperature, α is the thermal conductivity and Δ denotes the Laplacian operator. This equation, once discretized, has been applied to graph analysis numerous times including for supervised and semi-supervised classification of nodes. In the semi-supervised case, only a share of the nodes have a label that can be used for training [118, 7, 77].

Let $G = (V, E)$ be an undirected weighted graph, $n = |V|$ the number of nodes, A the adjacency matrix and D the degree matrix. Let $P = D^{-1}A$ be the transition matrix of a random walk on the graph. Let $\boldsymbol{T} \in \mathbb{R}^n$ be the vector of node temperatures, which is time dependent. The heat equation in discrete space can be written for each node as in Equation 6.1 or in a vector form as in Equation 6.2 where $L = D - A$ is the graph Laplacian (cf. Definition 23).

$$\forall i \in V, \frac{\partial \boldsymbol{T}_i}{\partial t} = \sum_{j \in V} A_{i,j}(\boldsymbol{T}_j - \boldsymbol{T}_i) \tag{6.1}$$

$$\frac{\partial \boldsymbol{T}}{\partial t} = -L\boldsymbol{T} \tag{6.2}$$

The Dirichlet problem consists in finding the equilibrium temperature vector $\boldsymbol{T}$ solving the heat equation 6.2 in the specific case where the temperature of a strict subset of nodes (denoted S for *seeds*) is fixed to a constant value at all time. Those are usually called boundary conditions. Let $s = |S|$ be the number of seeds and $\boldsymbol{T}^S \in \mathbb{R}^s$ the fixed temperatures. Up to a reordering of the nodes, $\boldsymbol{T}$ and P can be written in block form as in Equation 6.3 where $\boldsymbol{x} \in \mathbb{R}^{n-s}$ is the vector of unknown temperatures and Equation 6.4 where $Q \in \mathbb{R}^{(n-s)\times(n-s)}$ and $R \in \mathbb{R}^{(n-s)\times s}$.

$$\boldsymbol{T} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{T}^S \end{pmatrix} \tag{6.3}$$

$$P = \begin{pmatrix} Q & R \\ \cdot & \cdot \end{pmatrix} \tag{6.4}$$

A synthetic formulation of the Dirichlet problem is the following:

$$\begin{cases} \frac{\partial \boldsymbol{T}}{\partial t} = -L\boldsymbol{T} \\ \forall t, \forall s \in S, \boldsymbol{T}_s(t) = \boldsymbol{T}_s(0) \end{cases} \tag{6.5}$$

It can be proven that if G is connected, the Dirichlet problem 6.5 has a unique solution [28] mainly by showing that connectedness of G implies that $(I_{n-s} - Q)$ is invertible. It follows that the solution $\boldsymbol{x}$ is of the form $\boldsymbol{x} = (I_{n-s} - Q)^{-1}R\boldsymbol{T}^S$. Computing this exact solution can be costly. Instead, a very good approximation can be computed by iterating $\boldsymbol{x}^{(t+1)} = Q\boldsymbol{x}^{(t)} + R\boldsymbol{T}^S$. A proof of the convergence of this iterative method can be found in Appendix B for the more general case of directed graphs. A detailed formulation of the solver is proposed in Algorithm 6.

Solving a Dirichlet problem can easily be applied to binary node classification by using the nodes of the training set as seeds and setting their temperature either to 0 or 1 depending on their labels. The solution then gives a temperature for nodes that are not in the training set, which can be classified as 1 (resp. 0) if their temperature is above (resp. below) a given threshold. There are various strategies to choose the threshold: the naive one is to use $\frac{1}{2}$ but a more performing one is to use the average of the temperatures [33]. Figure 6.2 illustrates the Dirichlet binary classifier on the Karate Club graph [145, 33].

The generalization to multi-class classification is straightforward with a *one-vs-all* strategy: there are as many diffusion processes as there are distinct labels and the

Algorithm 6: Dirichlet solver

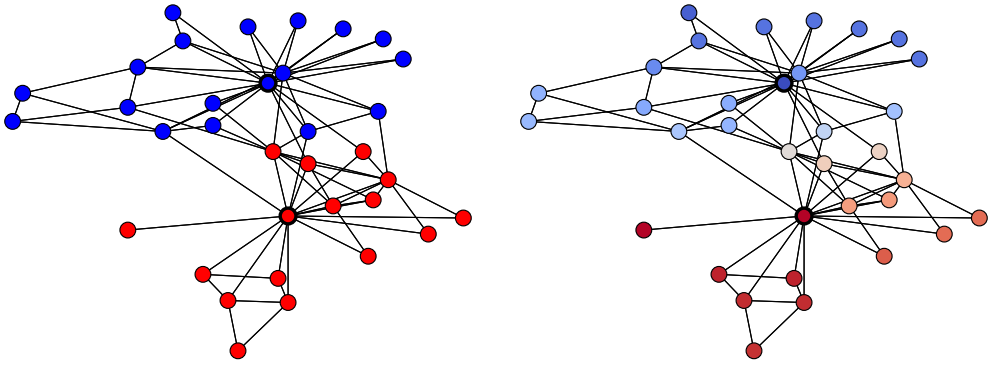
Input: $A \in \mathbb{R}^{n \times n}$ adjacency matrix of the graph

Input: $\boldsymbol{T} = \begin{pmatrix} (-1)^{n-s} \\ \boldsymbol{T}^S \end{pmatrix}$, with $\boldsymbol{T}^S \in \{0,1\}^s$ temperatures of the seeds

Input: K number of iterations

Output: $\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{T}^S \end{pmatrix}$, where $\boldsymbol{x} \in [0,1]^{n-s}$ vector of temperatures

$D \leftarrow A \cdot \mathbb{1}_n$;
$P \leftarrow D^{-1}A$;
$\boldsymbol{x} \leftarrow \text{mean}(\boldsymbol{T}^S)$;
$\boldsymbol{y} \leftarrow \boldsymbol{T}^S$;
for $k = 1, \ldots, K$ **do**
  $\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} \leftarrow P \cdot \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix}$;
  $\boldsymbol{y} \leftarrow \boldsymbol{T}^S$
return $\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{T}^S \end{pmatrix}$



(a) Ground truth. (b) Solution to the Dirichlet problem.

Figure 6.2: Binary classification of the Karate Club graph [145] with 2 seeds (indicated with a black circle). Red nodes have label 0, blue nodes have label 1. Visualization is extracted from [33].

seeds for each label are once set to 1 and the rest of the time set to 0. Eventually, temperatures are shifted so that the mean temperature of each diffusion is 0 and nodes are assigned the label that maximizes their temperature. This classifier is usually called the Dirichlet classifier and has no hyper-parameter. A detailed formulation is proposed in Algorithm 7.

Algorithm 7: Dirichlet Multi-class Classifier

Input: A adjacency matrix of the graph
Input: $\boldsymbol{y} \in (\{-1\} \cup C)^n$ the true labels (unknown nodes have true label -1)
Input: $t \in [0, 1]$ a threshold (default is 0)
Output: $\hat{\boldsymbol{y}} \in C^n$ the predicted labels

$\hat{\boldsymbol{y}} \leftarrow (-1)^n$
$T \leftarrow (0)^{|C| \times n}$; // matrix with all the class temperatures for each node.
for $c \in C$ **do**
  /* Binary classification for class c. */
  $\boldsymbol{T}^c \leftarrow (-1)^n$
  for $i = 0, \dots, n-1$ **do**
    if $\boldsymbol{y}_i = c$ **then**
      $\boldsymbol{T}^c_i \leftarrow 1$
    else if $y_i \neq -1$ **then**
      $\boldsymbol{T}^c_i \leftarrow 0$
  $\boldsymbol{T}^c \leftarrow binaryDirichlet(A, \boldsymbol{T}^c, 10)$
  $T_c \leftarrow \boldsymbol{T}^c - mean(\boldsymbol{T}^c)$; // $T_c \in [0,1]^n$ is the row of T corresponding to label c
/* Select the class of highest temperature respecting the threshold condition. */
for $i = 0, \dots, n-1$ **do**
  c_1, c_2 the two classes with highest temperature in $T_{:,i}$
  if $T_{c_1,i} > T_{c_2,i} + t$ **then**
    $\hat{\boldsymbol{y}}_i \leftarrow c_1$
return $\hat{\boldsymbol{y}}$

The Dirichlet classifier seems natural for entity typing in KGs as it can be applied in a semi-supervised setting while still using the entire adjacency matrix in the diffusion/learning process. This is particularly interesting in the case of KGs because usually many entities are not typed. For example, according the Wikidata official statistics[1], approximately 6% of the Wikidata entities had no type in 2020. The Dirichlet classifier can also be applied to a directed graph, with asymmetric adjacency matrix. Appendix B presents a proof of the existence and unicity of the solution to the Dirichlet problem along with a proof of the convergence of the iterative algorithm, both in the directed case. Though it seems appropriate to use the directed version for KGs, which are directed graphs, the experiments showed that the classifier works best with a symmetrized adjacency matrix, which is less sparse. Eventually, recall that KGs are defined by adjacency tensors, not matrices. There are various possibilities to apply Dirichlet classification to a KG. The simplest one would be to forget about the edge types (the relations) to turn the tensor into a matrix. This will be discussed in the experiments in Section 6.5.

6.2.2 Hierarchical Dirichlet classifier

An interesting feature of the Dirichlet classifier is that the temperature of each node can be interpreted as a confidence in the prediction. This classifier can then naturally be used in a local approach of HC such as the LCPN (cf. Section 6.1.2). Indeed, starting from the root class in the taxonomy, one chooses a sub-class and then moves down progressively, using the confidence measure as a stopping criterion in the traversal of

[1] https://www.wikidata.org/wiki/Wikidata:Statistics

the tree. It might be more interesting to stop the prediction on a more general type rather than making a mistake trying to specialize it.

We propose to call this algorithm Hierarchical Dirichlet and a detailed version is proposed in Algorithm 8. To the best of our knowledge, no publication has previously experimented with this hierarchical generalization of the Dirichlet classifier on graphs.

Generally speaking, the LCPN approach can be applied to any classifier that outputs a confidence metric in its prediction: in the experiments of Section 6.5, a hierarchical version of Euclidean nearest neighbor will be used as a baseline.

Algorithm 8: Hierarchical Dirichlet Classifier

Input: A adjacency matrix of the graph
Input: *child* a function returning the child classes of its argument
Input: $\mathcal{T}$ a graph traversal of the hierarchy
Input: *root* the most general class in the taxonomy
Input: $\boldsymbol{y} \in (\{-1\} \cup C)^n$ the true labels (unknown nodes have true label -1)
Input: $t \in [0, 1]$ a threshold
Output: $\hat{\boldsymbol{y}} \in C^n$ the predicted labels

$\hat{\boldsymbol{y}} \leftarrow (root)^n$
for $c \in \mathcal{T}$ **do**
  if $|child(c)| > 0$ **then**
    /* Then the current class is not a leaf. */
    /* Build the current ground truth vector. */
    $\boldsymbol{y}^c \leftarrow (-1)^n$
    for $i = 1..n$ **do**
      if $\exists \tilde{c} \in child(c) : \tilde{c} \preceq \boldsymbol{y}_i$ **then**
        $\boldsymbol{y}^c_i \leftarrow \tilde{c}$
      else
        $\boldsymbol{y}^c_i \leftarrow -1$
    /* Apply Dirichlet classifier to choose among the children of c. */
    $\hat{\boldsymbol{y}}^c \leftarrow Dirichlet(A, \boldsymbol{y}^c, t)$; // $\hat{\boldsymbol{y}}^c \in (\{-1\} \cup child(c))^n$
    /* Update predictions for nodes that were already assigned c. */
    for $i = 1..n$ **do**
      if $\hat{\boldsymbol{y}}_i = c$ *and* $\hat{\boldsymbol{y}}^c_i \neq -1$ **then**
        $\hat{\boldsymbol{y}}_i \leftarrow \boldsymbol{y}^c_i$
return $\hat{\boldsymbol{y}}$

6.3 Leveraging class taxonomy in gradient-based training with a hierarchical loss

This section introduces a hierarchical version of the binary cross-entropy loss to leverage the class taxonomy when training a node classifier such as a graph convolutional network.

6.3.1 Existing hierarchical losses

As explained in Chapter 3, neural networks and tensor models (e.g. KG embedding models) are usually trained using a gradient-based algorithm in order to minimize a training loss and assuming the set of classes comes with a taxonomy, a natural way to include it in the learning process is to make the training loss hierarchical.

Hierarchical losses for neural network training have not drawn much attention in the past and most of the literature focuses on hierarchical image classification. In 2019, Wu et al. [136] defined a metric that penalizes the model more, the further its prediction

falls from the ground truth in the taxonomy, relying on *ultrametric* trees in which each leaf is at the same distance from the root. The authors however express concerns on the comparative interest to usual flat loss. In 2019, Brust et al. proposed to encode existing hierarchies into a probabilistic model on the labels that is then used to derive a new label encoding [25]. The authors report some improvements in the overall accuracy but also a change in the training dynamics. In 2020, Bertinetto et al. expressed the cross-entropy of a prediction as a weighted sum of the cross-entropies of super-classes using an exponential discount factor linked to the depth in the hierarchy [8]. Eventually, Lucena proposed in 2022 a general framework to include any type of structure on the labels in entropy computations [82].

6.3.2 Hierarchical binary cross-entropy loss

When training tensor models on a classification task, a common loss is the Binary Cross-Entropy (BCE) expressed in Equation 6.6 where C is the set of possible classes, $\mathcal{T}$ is the set of training samples and given a such a sample i, $y_i \in C$ is the true class of i and $\boldsymbol{p}^{(i)} \in [0,1]^{|C|}$ is the estimated vector of probabilities that i belongs to each class.

$$-\sum_{i \in \mathcal{T}} \left(\log \boldsymbol{p}_{y_i}^{(i)} + \sum_{c \in C \setminus \{y_i\}} \log \left(1 - \boldsymbol{p}_c^{(i)}\right) \right) \tag{6.6}$$

Let us propose a hierarchical formulation of BCE that follows the principle presented by Bertinetto et al. in [8]: when computing the loss associated to a sample, the super-classes should be considered in order to weight how far the predicted one is from the true one.

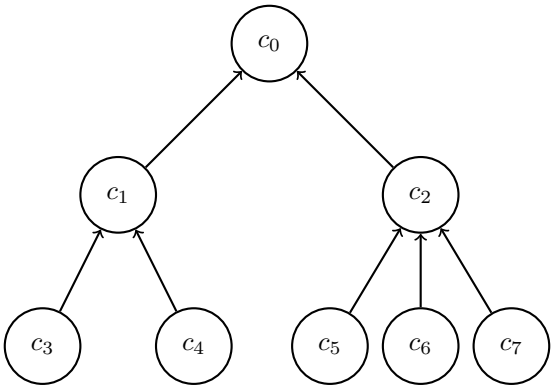
Precisely, the contribution to the loss of a sample i is given by Equation 6.7 where $\mathcal{A}(c)$ is the set of all super-classes of c up to the root (note that $c \in \mathcal{A}(c)$) and $\boldsymbol{p}_c^{(i)}$ is the estimated probability for node i to be of class c. Note that it has been assumed that taxonomies handled in this chapter are trees so the set $\mathcal{A}(c)$ is unequivocally defined.

$$\mathcal{L}_i = -\left(\sum_{c \in \mathcal{A}(y_i)} \log \boldsymbol{p}_c^{(i)} + \sum_{c \in C \setminus \mathcal{A}(y_i)} \log \left(1 - \boldsymbol{p}_c^{(i)}\right) \right) \tag{6.7}$$

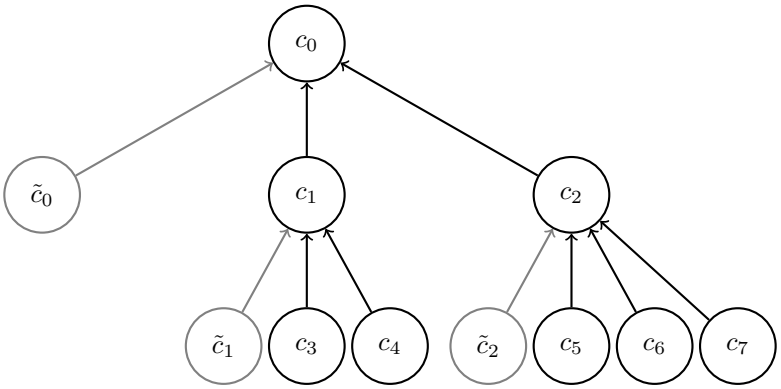
Equation 6.7 might include the probabilities computed by the model for any class of the taxonomy: leaves as well as inner-classes. Another constraint for the hierarchical BCE is that the probabilities of a class should be the sum of the probabilities of its sub-classes. This would result in an estimated probability distribution that is coherent with the taxonomy.

This last constraint is easily respected if the model only estimates probabilities on leaf-classes. The probability of inner-classes can then simply be computed by a bottom-up summation. In general however, some samples might be associated to inner-classes. It is moreover desirable to allow the model to predict a class high in the hierarchy with certainty rather than a leaf at random. This problem is solved by creating *artificial leaves* as sub-classes of the original inner-classes, as on Figure 6.3.

With artificial leaves, the classification model only estimates probabilities of leaf-classes and the probabilities of inner-classes are simply computed as the sum of the probabilities of their sub-classes. If the leaf probabilities sum to 1 (if the output of the classifier is normalized with a SoftMax function for example) then the probabilities smoothly sum to one when going up the taxonomy, resulting in a coherent probability distribution (recall that the taxonomy is supposed to be a tree).



(a) Original class hierarchy.



(b) New class hierarchy.

Figure 6.3: Toy example of the creation of artificial leaves $\tilde{c}_0$, $\tilde{c}_1$, $\tilde{c}_2$ as a child-classes of c_0, c_1 and c_2.

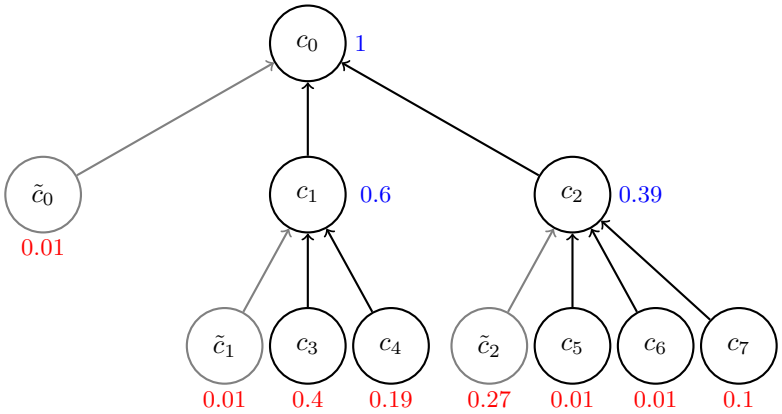


Figure 6.4: Example of an estimated probability distribution over the taxonomy presented in Figure 6.3. The estimated probabilities of the leaves are reported in red, while the hierarchically computed ones are in blue.

An example is proposed in Figure 6.4. The leaf probabilities that the model output are reported in red and the hierarchical summation is reported in blue. Such a distribution reflects that the model is confident in c_3 but, if asked, it would rather not choose between the sub-classes of c_2 (because $\boldsymbol{p}_{\tilde{c}_2}$ is larger than $\boldsymbol{p}_{c_5}$, $\boldsymbol{p}_{c_6}$ and $\boldsymbol{p}_{c_7}$).

Note that the proposed hierarchical BCE can be used to train any tensor model, though it will be studied in the specific case of graph convolutional networks in the following section.

6.3.3 Hierarchical graph convolutional network

Graph Convolutional Networks (GCNs) are a class of deep learning models designed to work on the topological structure of graphs the same way convolutional neural networks work on pixel structures in images. Precisely GCNs propagate information from one node to the other using what is called *convolution operators*. There exists a variety of such *operators* and depending on their nature, GCNs are categorized differently: *spatial* GCNs propagate information along edges, whereas *spectral* GCNs use spectral analysis to do so [24, 34, 67].The current section focuses on one of the most widespread spatial convolution operator, which was presented by Kipf and Welling in 2017 [67] and applied specifically to semi-supervised node classification, like the Dirichlet classifier previously introduced. In the rest of this chapter, GCN designates networks involving this specific operator.

Let $G = (V, E)$ be an undirected weighted graph, $n = |V|$ the number of nodes, A the adjacency matrix and D the degree matrix. Given a node i, let $\mathcal{N}(i)$ be the set of neighbors of i. Let $X \in \mathbb{R}^{n \times d}$ be the matrix of node features. A GCN is basically a succession of steps (corresponding to layers) that compute successive vector representations for each node, using the spatial convolution operator. Let $\boldsymbol{h}_i^{(\ell)}$ be a vector representation of node i at step ℓ, the successive representations are recursively defined following Equations 6.8 and 6.9 where W is a weight matrix and σ a non-liner function such as ReLU: $x \mapsto \max(0, x)$. Equation 6.8 is the aggregation step in which each node gets information from its neighbors and computes the average of their features. Following equation 6.9, the new hidden representation of each node is then computed by concatenating the averaged vector to the current representation of the node and feeding

the resulting vector to a fully connected network, the output of which is normalized.

$$\boldsymbol{h}_{\mathcal{N}(i)}^{(\ell+1)} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \boldsymbol{h}_j^{(\ell)} \tag{6.8}$$

$$\boldsymbol{h}_i^{(\ell+1)} = normalize\left(\sigma\left(W \cdot concat\left(\boldsymbol{h}_i^{(\ell)}, \boldsymbol{h}_{\mathcal{N}(i)}^{(\ell+1)}\right)\right)\right) \tag{6.9}$$

Each step corresponds to a layer in the GCN with the first representation of the nodes being set to the feature matrix $X \in \mathbb{R}^{n \times d}$. For classification, the dimension of the output layer is equal to the number of classes and the output is usually normalized between $[0, 1]$ using the SoftMax function of Equation 6.10 for example. This makes the output interpretable as probabilities for a node to belong to each class.

$$\boldsymbol{u} \in \mathbb{R}^n \mapsto \left(\frac{e^{\boldsymbol{u}_1}}{\sum_{i=1}^n e^{\boldsymbol{u}_i}}, \dots, \frac{e^{\boldsymbol{u}_n}}{\sum_{i=1}^n e^{\boldsymbol{u}_i}}\right) \tag{6.10}$$

The parameters of the model are limited to the weights of the fully-connected networks in each layer. Those are usually trained using gradient descent in order to find the configuration minimizing a loss such as the BCE (see Appendix A).

Interestingly, the Dirichlet classifier can be described as a particular case of a GCN. Let c be the number of classes, recall that the Dirichlet classifier is made of c binary classifiers in which, at each step, the nodes receive the average of their neighbor's temperature and the seeds have constant temperatures (either 0 or 1 depending on their label). This amounts to a GCN in which the input node features are a one-hot encoding of the known classes, the dimensions of the hidden representations are constant and the hidden fully connected layers (with non-linearity) are removed. Moreover, some nodes (the seeds) have constant hidden representations. The GCN is closer to the unconstrained problem of thermal diffusion without boundary conditions, the solution of which is a unique common temperature for all the nodes. A usual problem with GCNs is precisely the over-smoothing issue [27], which occurs when several steps of aggregation yield very close hidden representations for all the nodes.

We propose to train a hierarchical version of the GCN classifier using the hierarchical BCE introduced in the previous section.

6.4 The Self-Encoder model

The current section presents a flat model called the Self-Encoder. This is a simple and yet powerful model that can learn a geometry specific to the training samples that can be used on downstream tasks such as classification, as a kernel method.

The Self-Encoder is a neural network trained to guess the identity of each data sample. Given n data samples $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \in \mathbb{R}^d$, the objective of this encoder is to find a mapping $g : \mathbb{R}^d \to \{1, \dots, n\}$ such that $g(\boldsymbol{x}_i) = i$ for most samples $i = 1, \dots, n$. The output of the encoder is a probability distribution over the training samples that is computed using a latent representation of each sample.

6.4.1 Related work

Nearest Neighbors

A simple and yet fundamental method to solve tasks in machine learning is the proximity search. It relies on the intuition that close vectors in the feature space should have close properties (similar labels in classification and close values in regression). A straightforward application to supervised classification is the k-nearest neighbor (k-NN) algorithm [32] which assigns a label to a point by choosing the most present label among

its k closest neighbors. Usual similarity measures are the Euclidean metric or the cosine similarity but k-NN can also be applied with any similarity measure, such as the one learned by the proposed Self-Encoder model.

Kernel methods

Kernel methods are a class of ML algorithms that can work with similarity functions that are different from the usual Euclidean metric to measure the similarity between input vectors. Such a function is called a kernel and is usually defined by computing inner products involving all available pairs of data points.

One of the most famous kernel methods is the Support Vector Machine (SVM) [30]. Its description will give a better understanding of kernels. SVM is a supervised classification method that tries to find a boundary separating samples of different labels. In the simple case of binary classification, the dataset is said to be linearly separable if there exists an affine hyper-plane separating samples of both labels. In that case, the linear SVM *learns* the position of the hyper-plane by maximizing the margin, that is the distance of the hyper-plane to the closest training vector. In the case where the dataset is not linearly separable, the SVM can still be used with the help of a kernel that moves the training vectors into another space, possibly of different dimension, in which they are more likely to be separable. A usual example is the 2-dimensional case where positive samples are within a circle of radius 1 and negative samples are out of the circle. These points are not linearly separable in usual Cartesian coordinates but moving the points into the 1-dimensional space of the distance to the origin of the first space makes it separable.

The Self-Encoder is an unsupervised technique that learns a geometry of training samples to make them separable. By doing so, it learns a kernel that can compare new vectors to the training samples. A broader task is metric learning and aims at learning genuine distance functions (symmetric bilinear forms that verify the separability property and the triangle inequality). Importantly, the Self-Encoder is unsupervised making it theoretically applicable to any downstream task.

Auto-encoders

In its simplest form, an auto-encoder is a neural network model that learns a compact latent representation of data points with an encoder part and that is able to reconstruct them from this representation with a decoder part. The rationale is that a performing encoder should be able to extract the core discriminative features that define the samples of the dataset and this is measured by the ability of the decoder to reconstruct the original data from those features.

There is a large variety of application domains for auto-encoders and even more different structures. Let us cite for example variational auto-encoders that are used in a probabilistic framework for variational inference [66] or denoising auto-encoders that can be used for example to improve image resolutions by changing the reconstruction criterion [126] of manually noised data samples.

The output of an auto-encoder being a reconstructed version of the input vector, the measure of reconstruction loss is key in the training process. Unlike auto-encoders, in the case of the Self-Encoder, there is no need to engineer a good similarity measure between the original data sample and its reconstruction because the objective is simply to predict the identity of each data sample as an index in $\{1, \dots, n\}$ and not the data point in $\mathbb{R}^d$ itself.

6.4.2 Description of the Self-Encoder

Let $(\boldsymbol{x}_i)_{i\in\{1,\dots,n\}} \in \left(\mathbb{R}^d\right)^n$ be the set of training samples with d the dimension of the feature space and n the number of samples. The Self-Encoder is a Multi-Layer Perceptron

(MLP) with input dimension d and output dimension n, trained to predict the identity i of each data sample $\boldsymbol{x}_i$.

Hidden layers

The encoder consists of L hidden layers. Each layer $l = 1, \dots, L$ consists in an affine transformation followed by an activation function:

$$\begin{aligned}
\boldsymbol{h}^{(1)} &= \phi^{(1)}(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \\
\boldsymbol{h}^{(2)} &= \phi^{(2)}(W^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \\
&\vdots \\
\boldsymbol{h}^{(L)} &= \phi^{(L)}(W^{(L)}\boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)})
\end{aligned}$$

where $\phi^{(1)}, \dots, \phi^{(L)}$ are the activation functions, typically non-linear. The dimensions of the successive outputs, say $d_1, \dots, d_L$, are hyper-parameters. The weight matrices $W^{(1)}, \dots, W^{(L)}$ and the bias vectors $\boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(L)}$ must be learned.

Output layer

The output layer is a fully connected layer with input dimension d_L (the output dimension of the last hidden layer) and output dimension n. This affine transformation is followed by an activation function ϕ which is either a coordinate-wise sigmoid function $\phi : x \mapsto \frac{e^x}{1+e^x}$ or a SoftMax function (cf. Equation 6.10).

The output of the network is then a vector $\boldsymbol{p} = \phi(W\boldsymbol{h}^{(L)} + \boldsymbol{b}) \in [0,1]^n$ that can be interpreted as probabilities: the ith component $\boldsymbol{p}_i$ is the probability that the input $\boldsymbol{x}$ corresponds to the training sample $\boldsymbol{x}_i$. Observe that the probabilities sum to 1 only with the SoftMax function (with the sigmoid function, the probabilities are learned independently for each training sample i by the output layer). The weight matrix W and the bias vector $\boldsymbol{b}$ must be learned, together with the other parameters.

Loss function

In the following, f denotes the learned function of the network, mapping sample vectors $\boldsymbol{x} \in \mathbb{R}^d$ to probability vectors $\boldsymbol{p} \in [0,1]^n$. f is a parametric function and its parameters are the weight matrices $W^{(1)}, \dots, W^{(L)}, W$ and the bias vectors $\boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(L)}, \boldsymbol{b}$. Those are learned by minimizing the following BCE loss by gradient descent (cf. Appendix A):

$$\mathcal{L} = -\sum_{i=1}^{n} \left(\log f_i(\boldsymbol{x}_i) + \sum_{j \neq i} \log(1 - f_j(\boldsymbol{x}_i)) \right) \tag{6.11}$$

Interpretation

The Self-Encoder learns a latent representation of data, given by the last hidden layer, where the n training samples are linearly separable. It is the role of the output layer to find the hyperplanes (given by the weight matrix W and the bias vector $\boldsymbol{b}$) separating each training sample.

No hidden layer

Observe that the Self-Encoder can also be trained without any hidden layer. It then reduces to the output layer, i.e. a perceptron with input dimension d and output dimension n. Equivalently, the Self-Encoder then consists of n binary logistic regressions (for the sigmoid activation function) or a single multinomial logistic regression (for the SoftMax activation function).

Geometry

In both cases (with or without hidden layers), the Self-Encoder learns a specific similarity measure in the sense that it can predict the training samples that are the most similar to any new data sample $\boldsymbol{x}$. This measure depends on the distribution of the training samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ in the original space $\mathbb{R}^d$. Any sample $\boldsymbol{x} \in \mathbb{R}^d$ is said "*close*" to the training sample $\boldsymbol{x}_i$ if the corresponding predicted probability $\boldsymbol{p}_i = f_i(x)$ is close to 1.

6.4.3 Invariance property

The Self-Encoder is invariant to invertible affine transformations of the training data, as stated below.

Proposition 1. *Let f be the mapping learned by the encoder with training samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. For any invertible matrix $M \in \mathbb{R}^{d \times d}$ and vector $\boldsymbol{v} \in \mathbb{R}^d$, let $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n$ be the new training samples obtained by affine transformation $\boldsymbol{x} \mapsto M\boldsymbol{x} + \boldsymbol{v}$. The new mapping $\tilde{f}$ defined by:*

$$\forall \tilde{\boldsymbol{x}} \in \mathbb{R}^d, \quad \tilde{f}(\tilde{\boldsymbol{x}}) = f(M^{-1}(\tilde{\boldsymbol{x}} - \boldsymbol{v}))$$

minimizes the cross-entropy loss (6.11) for the training samples $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n$. Both encoders are related through the affine transformation:

$$\forall \boldsymbol{x} \in \mathbb{R}^d, \quad f(\boldsymbol{x}) = \tilde{f}(M\boldsymbol{x} + \boldsymbol{v}).$$

In other words, if the training data are the same up to some invertible affine transformations, so are the mappings learned by the encoder.

Proof. The mapping $\tilde{f}$ is the same encoder as f except for the first hidden layer, whose output $\tilde{\boldsymbol{h}}^{(1)}$ for the input $\tilde{\boldsymbol{x}}$ is given by:

$$\begin{aligned}
\tilde{\boldsymbol{h}}^{(1)}(\tilde{\boldsymbol{x}}) &= \boldsymbol{h}^{(1)}\left(M^{-1}(\tilde{\boldsymbol{x}} - \boldsymbol{v})\right) \\
&= \phi^{(1)}\left(W^{(1)}(M^{-1}(\tilde{\boldsymbol{x}} - \boldsymbol{v})) + \boldsymbol{b}^{(1)}\right) \\
&= \phi^{(1)}\left(W^{(1)}M^{-1}\tilde{\boldsymbol{x}} + \boldsymbol{b}^{(1)} - W^{(1)}M^{-1}\boldsymbol{v}\right) \\
&= \phi^{(1)}(\tilde{W}^{(1)}\tilde{\boldsymbol{x}} + \tilde{\boldsymbol{b}}^{(1)}),
\end{aligned}$$

for the weight matrix and bias vector:

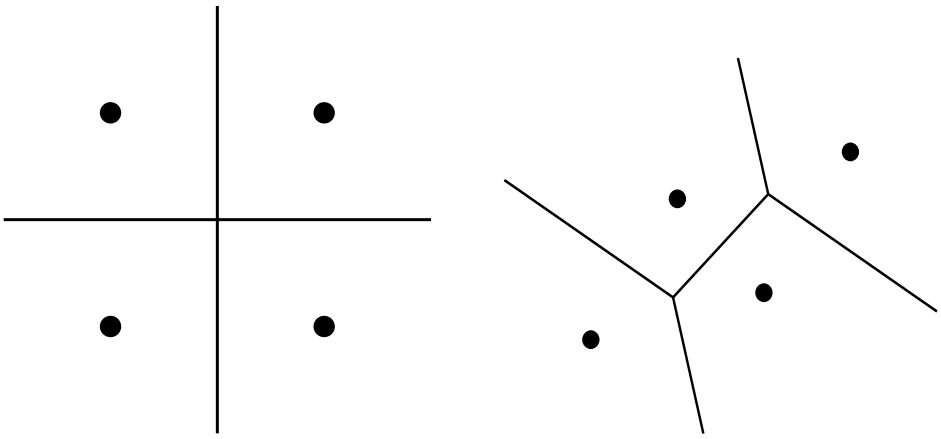
$$\begin{aligned}
\tilde{W}^{(1)} &= W^{(1)}M^{-1} \\
\tilde{\boldsymbol{b}}^{(1)} &= \boldsymbol{b}^{(1)} - W^{(1)}M^{-1}\boldsymbol{v}.
\end{aligned}$$

The corresponding binary cross-entropy loss is minimized, given that:

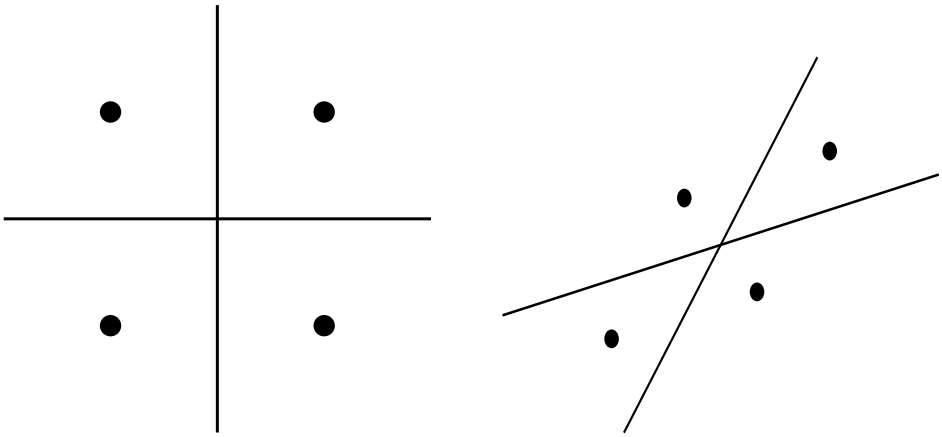
$$\begin{aligned}
\mathcal{L} &= -\sum_{i=1}^{n}\left(\log \tilde{f}_i(\tilde{\boldsymbol{x}}_i) + \sum_{j \neq i}\log(1 - \tilde{f}_j(\tilde{\boldsymbol{x}}_i))\right) \\
&= -\sum_{i=1}^{n}\left(\log f_i(\boldsymbol{x}_i) + \sum_{j \neq i}\log(1 - f_j(\boldsymbol{x}_i))\right)
\end{aligned}$$

□

This invariance property is a clear difference with the usual Euclidean distance. To illustrate this, Figure 6.5 shows the Voronoi diagram (regions formed by the nearest neighbors) associated with $n = 4$ points of $\mathbb{R}^2$ for the Self-Encoder and for the Euclidean distance. On the left, the 4 points form a square and the Voronoi diagrams coincide,



(a) Euclidean distance



(b) Self-Encoder

Figure 6.5: Impact of a linear transformation on the Voronoi diagram (regions of nearest neighbors) formed by $n = 4$ points in $\mathbb{R}^2$.

by symmetry. On the right, a linear transformation is applied; the Voronoi diagram is obtained by the same linear transformation for the Self-Encoder, while it changes completely for the Euclidean distance.

This invariance property is handy as it simplifies the pre-processing steps, which can become tedious and that usually have an impact on the performances of many classifiers. This will be showed later in the experiments for the Euclidean k-NN classifier.

6.4.4 Categorical features

We claim that the Self-Encoder robustly handles categorical features, in the sense that it does not depend on the number of bits they are encoded on, unlike Euclidean Nearest Neighbor (NN).

Given $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$ training samples with categorical features in $\{0,1\}^d$, data redundancy in the features does not modify the optimum of the function. If there is a pair (k_1, k_2) such that $\forall i, \boldsymbol{x}_{k_1}^{(i)} = \boldsymbol{x}_{k_2}^{(i)}$ then the contributions to the loss involving these features is duplicated and they should result in the same corresponding weights in the input layer, which should not impact the learned geometry. If there is a pair (k_1, k_2) such that $\forall i, \boldsymbol{x}_{k_1}^{(i)} = 1 - \boldsymbol{x}_{k_2}^{(i)}$ then the k_1-th feature is a invertible affine transformation of the k_2-th feature and the invariance property of Proposition 1 tells us that the first case of redundancy applies.

Let us illustrate on a simple example how the similarity measure learned by a Self-Encoder can outperform the usual Euclidean metric on a nearest-neighbor search by handling categorical features differently.

$$X_1 = \begin{pmatrix} \boldsymbol{x}_1^{(1)} \\ \boldsymbol{x}_1^{(2)} \\ \boldsymbol{x}_1^{(3)} \\ \boldsymbol{x}_1^{(4)} \\ \boldsymbol{x}_1^{(5)} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \tag{6.12}$$

$$X_2 = \begin{pmatrix} \boldsymbol{x}_2^{(1)} \\ \boldsymbol{x}_2^{(2)} \\ \boldsymbol{x}_2^{(3)} \\ \boldsymbol{x}_2^{(4)} \\ \boldsymbol{x}_2^{(5)} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{6.13}$$

Let X_1 and X_2 be defined in Equations 6.12 and 6.13. The only difference between the two set of samples is that the binary feature of the first binary column of X_1 is encoded on two bits in X_2 in the first two columns. It is reasonable to expect that a classifier trained on X_1 and fed with $\bar{\boldsymbol{x}}_1 = \begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix}$ makes the same decision as a classifier trained on X_2 and fed with $\bar{\boldsymbol{x}}_2 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \end{pmatrix}$. This is not the case for a Euclidean nearest-neighbor as the closest vector from $\bar{\boldsymbol{x}}_1$ in X_1 is $\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix}$ and the closest ones from $\bar{\boldsymbol{x}}_2$ in X_2 are $\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \end{pmatrix}$, $\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \end{pmatrix}$. On the other hand, the Self-Encoder returns $\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix}$ as the most similar to $\bar{\boldsymbol{x}}_1$ and $\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \end{pmatrix}$ as the most similar to $\bar{\boldsymbol{x}}_2$.

6.4.5 Sampling

Under the reasonable assumption that the dimensions of the hidden layers do not scale with n, the time complexity is that of the output layer: $O(n^2)$ for training and $O(n)$ for evaluation. The training time complexity is higher than the n-linear time complexity of Euclidean k-NN. However, training is only done once and then the trained model can be used in $O(n)$ for similarity search. Moreover, the natural way to implement a MLP model is to use machine learning frameworks such as PyTorch [99] or TensorFlow [1], which natively support GPU acceleration and highly reduce the computation time.

To control the memory usage and time complexity of the approach and in order to improve the performance, we propose to use a simple sampling strategy, where a random subset of the training set is selected, thus reducing the space and time complexity of the model. Given a set of training samples $X = (\boldsymbol{x}^{(i)})_{i \in \{1,\dots,n\}}$, sampling generates a new training subsets of size s by randomly sampling vectors from X. The model is then trained on the new training subset. Experiments show that that the performance of the sampled model is comparable to the complete one.

6.4.6 Self-Encoder for flat classification

Experimental setting

To evaluate the quality of the similarity measure learned by the proposed Self-Encoder, we apply it to a flat classification task with k-NN method (with $k = 3$). The Self-Encoder k-NN is simply called Self-Encoder for simplicity. It is tested with and without a hidden layer and in normal and sampling modes. Training uses early stopping and learning rate decay (cf. Appendix A).

The performances are compared to those of three classification baselines: the Euclidean k-NN with $k = 5$, the linear Support Vector Machine (SVM) and the one-vs-all logistic regression.

	# samples	Feature dimension	# classes
Breast Cancer [84]	699	9	2
Digits [3]	1,797	64	10
Ecoli [89]	336	7	8
German credit [58]	1,000	24	2
German credit (categorical) [58]	1,000	20	2
Glass [51]	214	9	6
Ionosphere [109]	351	34	2
Iris [46]	150	4	3
Liver [81]	345	6	2
Wine [13]	178	13	3

Table 6.2: Dataset details

Ten datasets that are recurrent in the machine learning literature were selected for the experiments. They are all available from the UCI repository[2] and descriptive figures can be found in Table 6.2 along with references. Among these, the German Credit dataset comes in two versions: one with numerical features and another one with 13 out of 20 features being categorical. The only pre-processing applied to all the datasets is the conversion of categorical features into a one-hot encoding.

Some hyper-parameters are fixed: the learning rate decay is set to 0.995, the size of the hidden layer is fixed to 20 and for the sampling mode, the number of visible samples is fixed to 100. The learning rate is chosen according to a log-uniform distribution between 0.001 and 2, using the Bayesian optimization library `hyperopt`[3]. The library is also used to choose the normalization function between sigmoid and SoftMax.

For each classification model and each dataset, the reported metric is the accuracy. It is measured using 5-fold cross validation.

Results

All results are reported in Table 6.3. An obvious conclusion is that in both normal and sampling settings, the Self-Encoder performs better than the baselines. A second conclusion is that the Self-Encoder in the sampling framework performs comparably or better than the other baselines in the normal setting. This is reassuring because the Self-Encoder might need to be applied in sampling mode while other lighter models have access to all the samples.

To measure the impact of the invariance feature, the performance of the Self-Encoder has also been compared to the performance of the Euclidean k-NN algorithm on normalized numerical datasets. The results are shown in Table 6.4. As expected, normalization improves the score of the Euclidean k-NN in most cases but not up to the score of the Self-Encoder.

In conclusion, the Self-Encoder is an unsupervised method that learns a similarity measure specific to the training data that can be used for downstream tasks such as classification. Its primary objective being to separate samples from one another, it is expected that its predictions should comply correctly with the hierarchy if the dataset is rich enough.

[2] https://archive.ics.uci.edu/ml/datasets.php
[3] https://hyperopt.github.io/hyperopt

	k-NN		MLP		Logistic		SVM	
Breast cancer	**0.967**	0.009	0.96	0.007	0.963	0.003	0.96	0.003
Digits	**0.983**	0.004	0.934	0.013	0.965	0.004	**0.983**	0.002
Ecoli	**0.869**	0.021	0.762	0.016	0.759	0.029	0.798	0.019
German credit	0.684	0.012	0.741	0.026	0.738	0.012	**0.754**	0.012
German credit cat	0.716	0.027	0.717	0.023	0.733	0.027	**0.736**	0.023
Glass	**0.616**	0.066	0.439	0.099	0.579	0.046	**0.616**	0.06
Ionosphere	0.835	0.032	**0.903**	0.033	0.855	0.039	0.838	0.03
Iris	0.953	0.05	0.913	0.129	0.967	0.03	**0.987**	0.027
Liver	**0.693**	0.025	0.609	0.098	0.684	0.063	0.687	0.057
Wine	0.714	0.046	0.321	0.1	0.966	0.021	**0.977**	0.011

(a) Classification accuracies of the baseline methods.

	SE		SE hidden		Best Baseline	
Breast cancer	0.968	0.007	**0.976**	0.011	0.967	0.009
Digits	**0.983**	0.012	0.655	0.117	**0.983**	0.004
Ecoli	**0.911**	0.028	0.899	0.021	0.869	0.021
German credit	**0.768**	0.011	0.749	0.007	0.754	0.012
German credit cat	**0.768**	0.025	0.756	0.02	0.736	0.023
Glass	**0.794**	0.033	0.766	0.041	0.616	0.06
Ionosphere	0.94	0.023	**0.963**	0.031	0.903	0.033
Iris	**1.0**	0.0	0.993	0.013	0.987	0.027
Liver	0.759	0.02	**0.768**	0.016	0.693	0.025
Wine	0.736	0.071	0.871	0.116	**0.977**	0.011

(b) Classification accuracies of the Self-Encoder. The score of the best baseline is recalled for comparison.

	SE		SE hidden		Best Baseline	
Breast cancer	0.960	0.006	**0.970**	0.008	0.967	0.009
Digits	0.853	0.019	0.528	0.161	**0.983**	0.004
Ecoli	0.878	0.036	**0.899**	0.025	0.869	0.021
German credit	0.741	0.012	0.750	0.008	**0.754**	0.012
German credit cat	0.742	0.011	**0.750**	0.011	0.736	0.023
Glass	0.766	0.03	**0.771**	0.027	0.616	0.06
Ionosphere	0.897	0.028	**0.946**	0.017	0.903	0.033
Iris	**1.0**	0.0	0.993	0.013	0.987	0.027
Liver	**0.762**	0.024	0.760	0.007	0.693	0.025
Wine	0.702	0.04	0.848	0.073	**0.977**	0.011

(c) Classification accuracies of the Self-Encoder in sampling mode (with 100 visible samples chosen uniformly at random). The score of the best baseline is recalled for comparison.

Table 6.3: Classification accuracies for the Self-Encoder along with other baselines on ten usual datasets. Performances are measured using a 5-fold cross validation mechanism. Best results are in bold. German credit cat is the version of the German credit dataset with categorical features.

	k-NN normalized	k-NN	SE
Breast cancer	0.960	**0.967**	**0.976**
Digits	0.970	**0.983**	**0.983**
Ecoli	0.866	**0.869**	**0.911**
German	**0.693**	0.684	**0.768**
Glass	**0.645**	0.616	**0.794**
Ionosphere	**0.835**	**0.835**	**0.963**
Iris	**0.960**	0.953	**1.0**
Liver	0.609	**0.693**	**0.768**
Wine	**0.966**	0.714	**0.871**

Table 6.4: Classification accuracy of Euclidean k-NN on normalized datasets and Euclidean k-NN and Self-Encoder on raw datasets. This comparison is only reported on numerical datasets. Two best scores for each dataset are in bold.

6.5 Experiments

This section describes the experimental setup used to compare the models introduced in the three previous sections on HC. Two different datasets are used, Wikivitals+ and WDV5, which were already introduced in Chapter 5. Both datasets come with a taxonomy on the target classes.

All the metrics reported were computed using 5-fold cross-validation method and the training hyper-parameters were chosen by grid-search on a random 20% subset of the training set.

6.5.1 Datasets

The experiments involve two datasets. Both are made of a graph where the nodes have a feature vector and a class. Classes are hierarchically organized in a taxonomy. Descriptive figures of both datasets can be found in Table 6.5[4].

- Wikivitals+[5] is an extraction of the level 5 of Wikipedia vital articles[6] and the hyperlinks linking them. The labels are hierarchical and come from the Wikipedia categories. In the experiments, the hierarchy is manually cut to depth 3. The node features are made by a bag-of-word representation of the page descriptions. Initially, 85,512 words are used but for lighter computations, only 500 hundred words are kept during training. The words with the lowest TF-IDF scores on the training set are removed. A proper definition of the TF-IDF weighting scheme [107] can be found in Appendix C.
- WDV5 is a subset of Wikidata containing the facts linking entities corresponding to the Wikipedia pages of Wikivitals+. As explained in Chapter 2, typing in Wikidata is based on the *P31: instance of* relation and the taxonomy is not much constrained. For simplicity, entities in WDV5 have been typed using the YAGO4 taxonomy which is simpler [101]. However, approximately 26% of the entities in WDV5 are not present in YAGO4, because the latter is built by discarding types with too few entities (and the corresponding entities). This is not a problem though, because the Dirichlet and GCN classifiers work in a semi-supervised setting as well. WDV5, which is a KG, is turned into a graph simply by filling the adjacency matrix with the facts deprived of the relation. Node features are made of a *bag-of-relations* representation of the relation signatures of entities. For example,

[4] WDV5 is larger in the current chapter as it was rebuilt since the work presented in Chapter 5 and it now includes more entities.

[5] https://netset.telecom-paris.fr/pages/wikivitals+.html

[6] https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Level/5

	Wikivitals+	WDV5
# nodes	45,179	42,679
# edges	3,946,850	260,350
# labeled nodes	45,179	31,493
# labels	540	1,688
Taxonomy depth	3	7
Feature dimension	85,512	648

Table 6.5: Details of Wikivitals+ and WDV5.

a fact $\langle h, r, t \rangle$ yields an edge between h and t in the graph and a 1 entry in the feature vectors of h and t.

6.5.2 Classification models

The following models are tested:

- Dirichlet classifier and the hierarchical version
- GCN classifier trained with usual BCE (noted GCN)
- GCN classifier trained with hierarchical BCE (noted hGCN)
- k-NN using the similarity measure of a Self-Encoder in sampling mode (10,000 visible samples and $k = 3$)

As it was introduced, the Dirichlet classifier does not support node features. The simplest way to include them in the diffusion process and make all models run exactly on the same input data is to add as many *feature* nodes in the graph as there are features and add corresponding edges between the original nodes and the feature nodes.

Two baselines are also included: the usual Euclidean k-NN algorithm with $k = 5$ and a hierarchical version of it, following the same LCPN method as the hierarchical Dirichlet classifier. The predictions of the k-NN algorithm are weighted by the share of each class among the k nearest neighbors.

6.5.3 Performance metrics

Three metrics are reported in order to measure the performances of each model on a specific criteria:

- the accuracy measures the ability of a model of being right. There is no measure of the *gravity* of the mistakes relatively to the hierarchy.
- the hierarchical f_1-score introduced in Section 6.1.3 is expected to measure the performance of the model in complying with the hierarchy.
- the projected accuracy is defined as the accuracy computed after projecting the true and predicted labels on a common depth of the taxonomy. For example if a node should be classified as *Thing/Human/Athlete*, the new true label at depth 1 should be *Thing/Human*. This is supposed to measure how well a model performs at a given depth in the hierarchy.

6.6 Results

All the results are reported in Table 6.6. Several conclusions can be drawn from them.

First, the results of the hierarchical BCE are conclusive. Even if it does not significantly impact the accuracy results of the GCN model, it improves the other hierarchical scores: up to 7% increase in the projected accuracy on level 1 on WDV5. The LCPN approach on the other hand has more mixed results. Despite often improving the hierarchical metrics, it always decreases the accuracy.

Comparing the performances of the flat Dirichlet classifier to its hierarchical version and the performances of the GCN classifiers trained with flat and hierarchical losses gives the sense that involving the taxonomy in the training procedure does not improve their overall propensity of predicting the right labels (measured by the accuracy). It can however modify the way predictions are made and improve the compliance of the predictions with the taxonomy. This agrees with the concerns expressed by [136] on the overall interest of hierarchical loss as well as with a 1998 technical note [88] by Mitchell, which states that under reasonable assumptions, the naive Bayes classifier is in fact equivalent in term of accuracy to its obvious hierarchical version. Though this note specifically deals with naive Bayes classifier, the intuition it conveys is confirmed by the current experimental results.

The results of the various models on WDV5 also question the choice of graph approaches to solve the classification task. In Wikivitals+, the features are an encoding of descriptive words, while in WDV5, they are an encoding of neighboring relations. The aggregation of such features accross neighboring nodes is questionable. An interesting follow-up would be to experiment the training of a multi-layer perceptron classifier with the proposed hierarchical BCE loss. Such a model would not take into account the graph structure of the nodes and could help quantify the signal spread in the features and its impact in extracting the taxonomy.

The results also lead to a criticism of the hierarchical f_1-score, which can sometimes be misleading. For example, the Self-Encoder on WDV5 performs better than other models in term of hierarchical f_1-score and yet this is not reflected in the projected accuracies. A possible explanation is that the metric gives more weight to samples with labels down in the hierarchy, explaining why the phenomenon is more striking with WDV5, which has a deeper taxonomy than Wikivitals+.

Eventually, the results of the Self-Encoder are also rich in possible interpretations. The model performs better than the Euclidean k-NN and Dirichlet classifier on Wikivitals+ on all the metrics. It also gives reasonable results on WDV5 by performing better for example than the two GCN models. The underlying intuition is that a flat classifier should be able to implicitly comply with the taxonomy directly by mining patterns from the node features. For example, some features such as a *P1532: country for sport* relation might be instrumental in arbitrating if a node is either a *Thing/Human* or a *Thing/Human/Athlete*. Obviously, this is conditional to having enough data points to make the classes distinguishable in the feature space.

An interesting possible follow-up would be to apply the similarity measure learned by the Self-Encoder to do clustering on the labels and to compare it to the results of existing hierarchical clustering methods. This experiment is however difficult to conduct because hierarchical clustering usually outputs a dendrogram that needs to be cut and aligning the clusters of a cut dendrogram is a tedious task, especially if the hierarchy is not well-balanced.

	Wikivitals+	WDV5
k-NN	0.441 ± 0.007	**0.731** ± 0.002
Hierarchical k-NN	0.407 ± 0.005	0.608 ± 0.003
Dirichlet	0.514 ± 0.004	0.713 ± 0.003
Hierarchical Dirichlet	0.408 ± 0.002	0.685 ± 0.003
GCN	0.650 ± 0.004	0.654 ± 0.002
hGCN	**0.654** ± 0.004	0.653 ± 0.005
Self-Encoder sampling	0.552 ± 0.003	0.655 ± 0.005

(a) Accuracy

	Wikivitals+	WDV5
k-NN	0.648 ± 0.013	0.847 ± 0.002
Hierarchical k-NN	0.657 ± 0.005	0.840 ± 0.001
Dirichlet	0.701 ± 0.003	0.856 ± 0.001
Hierarchical Dirichlet	0.685 ± 0.002	0.870 ± 0.001
GCN	0.798 ± 0.003	0.856 ± 0.003
hGCN	**0.811** ± 0.003	0.821 ± 0.002
Self-Encoder Sampling	0.736 ± 0.002	**0.895** ± 0.002

(b) Hierarchical f_1-score

	Wikivitals+	WDV5
k-NN	0.649 ± 0.033	0.847 ± 0.002
Hierarchical k-NN	0.678 ± 0.004	0.858 ± 0.003
Dirichlet	0.698 ± 0.006	0.848 ± 0.003
Hierarchical Dirichlet	0.705 ± 0.003	**0.866** ± 0.002
GCN	0.783 ± 0.004	0.686 ± 0.004
hGCN	**0.849** ± 0.003	0.759 ± 0.006
Self-Encoder Sampling	0.755 ± 0.003	0.776 ± 0.006

(c) Projected accuracy on level 1.

	Wikivitals+	WDV5
k-NN	0.500 ± 0.009	0.768 ± 0.003
Hierarchical k-NN	0.472 ± 0.002	0.715 ± 0.001
Dirichlet	0.592 ± 0.005	**0.775** ± 0.003
Hierarchical Dirichlet	0.553 ± 0.002	0.769 ± 0.002
GCN	0.730 ± 0.005	0.659 ± 0.004
hGCN	**0.743** ± 0.005	0.695 ± 0.005
Self-Encoder Sampling	0.635 ± 0.002	0.708 ± 0.006

(d) Projected accuracy on level 2.

Table 6.6: Performance metrics of the different models. Metrics are measured by a random 5-fold cross validation.

Chapter 7

Conclusion

In this thesis we have tackled several problems related to knowledge graph automatic completion.

First, Chapter 2 introduced the concepts of knowledge base and knowledge graph and listed some of the existing public knowledge bases. Then the task of knowledge graph automatic completion was defined in Chapter 3 before reviewing existing machine learning techniques as well as symbolic and hybrid methods. A conclusion drawn from this review is that the field lacks diversity in the datasets used and simple models might constitute a good practical choice if properly trained and even more if combined with symbolic methods to exploit ontologies.

In Chapter 4, TorchKGE was introduced as a Python library specifically designed to provide users with an efficient and user-friendly framework to implement and experiment with machine learning methods for KG completion. A key feature of the library is a very efficient evaluation module that relies on the vectorization of the computations in inference mode. This is useful to properly train the models. In the future, the library could benefit from a further development of the tools related to entity typing. To improve its adoption by the community, its simple usage should be put forward to stand out from the competition of other existing libraries. We shall also pay particular attention to the empowerment of the community of contributors.

Then Chapter 5 mainly proposed a method to enrich Wikidata by automatically assigning a semantic relation to existing Wikipedia hyperlinks. This method relies on a new negative sampling technique used to train simple models like TransE combined with an inference scheme that uses both the predictions from the embeddings and the entity types. Another contribution of this work is the new WDV5 dataset. This chapter however raises the question of trust in machine learning methods. It is unclear how such methods are likely to be adopted on real-world projects and importantly what are the conditions to their adoptions. A first step to their adoption certainly is as suggestion methods that still require a human verification. This question goes beyond the specific topic of knowledge graph completion.

Eventually Chapter 6 dived into the general problem of hierarchical classification. This is motivated by entity typing, which is another way of completing knowledge graphs. After introducing related works and definitions, a hierarchical formulation of the well-known binary cross-entropy loss was proposed to use available taxonomies as an inductive bias to train tensor-based classification models. It was then used to train a graph convolution network on a hierarchical classification task. Another contribution of this chapter is the proposed Self-Encoder model. That is an unsupervised machine learning model that is trained to learn a separable representation of training vectors and by doing so, it learns a similarity measure that can be used for downstream tasks such as classification. Experiments were conducted to measure the impact of an a priori

knowledge of the hierarchy on the performance of classifiers. The conclusion is that a model trained with knowledge of the hierarchy might not make fewer mistakes but those mistakes should be more compliant with the hierarchy. On the other hand, the unsupervised Self-Encoder method, showed that a flat model should be able to comply with the hierarchy simply using patterns the features, conditionally to the dataset being rich enough.

In this last chapter, the similarity metric from the Self-Encoder model has been applied to entity typing with a k-NN proximity search. It would be interesting to also apply it to the two other tasks tackled in the thesis: link and relation prediction. An obvious issue with these is that the proximity search needs to involve a third object and it is not clear how. We can however imagine other completion tasks, for example, similar entities might have common outgoing relations, defining their properties. For example, human entities should have a place of birth. The Self-Encoder could then be used to determine which properties are missing for an entity by looking at the ones of its *close* neighbors. This task is linked to the broader problem of ontology design that can require specific properties in the definition of typed entities in knowledge bases.

Another interesting subject that was not investigated in the last chapter is the design of taxonomies. This is a research topic on its own and the motivations are concrete. The taxonomy of Wikidata for example, though semantically rich, is difficult to process automatically because of its sparseness and lack of constraints. This is why the taxonomy from YAGO4 was used to type the entities of WDV5 in Chapter 6. Machine learning and symbolic methods are likely to perform well on class de-duplication and clustering or more generally on taxonomy improvement.

Another further application of the contributions presented in this thesis is the knowledge base correction task. As it was evoked in the introduction, this task is complementary to the automatic completion: if an algorithm can predict likely facts, then it should be able to rate the existing ones that are very unlikely. A motivation for the works of Chapters 5 and 6 was the very large number of possible candidate triples to predict new facts. It seemed judicious to reduce this set by mining possible candidates from other data sources (Wikipedia hyperlinks) or by focusing on a specific relation (types). When it comes to correction on the other hand, the list of possible erroneous candidates is simply made of the existing facts making it possible to review all of them.

To conclude, let us reflect on the recent trendy ChatGPT[1] tool proposed by the company OpenAI. Independently of the large media coverage it received, this tool might truly represent an important step in the field of artificial intelligence. Up to now, large language models had successfully been applied to several tasks but they remained linguistic models, lacking an ordered sense of knowledge. A language model indeed is *only* trained to compute a probability distribution on a set of words given a context. Certainly, existing patterns, in lexical fields for example, often resulted in reasonable generated sentences but hands-on experiments with models such as GPT-3 [23] quickly showed that the model had no understanding of the concepts and entities that were involved in its sentences. This is the field of knowledge bases and reasoning. ChatGPT however has shown that a tuned large language model can result in surprising discussion capabilities. There are still many unknowns about the true functioning of the model, which has not been published yet, but experiments show that when it comes to factual discussions it is able to answer successive queries very accurately and with surprising temporal coherence. There are still some major limitations such as the very poor logic and reasoning capabilities and the impossibility to quote sources of facts. It can however be considered as a proof-of-concept, showing that large language models can learn and store knowledge in a diffuse neural way, at the opposite of the structural approach of knowledge bases, and probably closer to the way our brains work. This breakthrough provides a new perspective on the fundamental question of the storage and processing of knowledge by machines.

[1] https://chat.openai.com/chat

Appendix A

Usual neural network training techniques

This appendix briefly introduces some training techniques used in this thesis and commonly applied to neural networks. The list obviously is not exhaustive and pointers to more detailed sources are provided.

A.1 Neural network training

Training tensor-based models, such as neural networks, is usually done using gradient-based techniques. An extensive presentation and analysis of these can be found in Chapter 8 of [52].

Let f_W be a parametric function with W its parameters. Let $(\boldsymbol{x}_i, y_i)_{i\in\{1,\dots,n\}}$ be a set of training data points with target values. f_W can for example be the function of a neural network or the scoring function of a KG embedding model. The training process comes down to finding the set of parameters W that minimizes a loss function when evaluated in the training set. The loss function should reflect how far the output of f_W is from the target values when evaluated on the training points. Other common names are cost function, error function, objective function, or criterion.

Functional analysis states that under proper regularity conditions, a function can be minimized by moving small steps in the direction opposite to the gradient. This is the underlying principle applied to train almost all the tensor-based models, such as neural networks. Algorithm 9 shows a general formulation of the gradient descent training process where $\nabla_W \mathcal{L}(W)$ is the gradient of $\mathcal{L}$ with respect to the parameters W.

Algorithm 9: Training a model by gradient descent.

Input: f_W the model function
Input: γ the learning rate
Input: ℓ the loss function
Output: W^* the optimal set of parameters

randomly initialize W
for a fixed number of epochs **do**
  $\mathcal{L}(W) \leftarrow \sum_{i=1}^{n} \ell\left(y_i, f_W(\boldsymbol{x}_i)\right)$
  $W \leftarrow W - \gamma \nabla_W \mathcal{L}(W)$
return W

In general, the gradient is not computed for the loss over all the training points but rather for intermediate smaller losses over smaller batches of training points (batch and

mini-batch gradient descent). Another possibility is to compute it on training points chosen at random (stochastic gradient-descent) (See Section 8.1.3 of [52]).

A common gradient-descent based algorithm used for training tensor-based models is Adam [65]. Its conception relies on the intuition that the learning rate should not be constant during the whole process. Indeed, it should be larger at the beginning to explore large parts of the space of possible parameters but become smaller toward the end to end up very close to the optimum. One of the hyper-parameters of the algorithm is the learning-rate decay, which states how fast the learning rate should decrease. Adaptive learning rates are presented in Section 8.5 of [52].

A.2 Regularization

As explained in Chapter 3, ML models in general are judged on the ability to generalize their performance to previously unseen data points. If a model generalizes poorly, it is said to overfit its training data. There is a variety of techniques to avoid overfitting. The current appendix presents three common ones but more can be found in Chapter 7 of [52].

A.2.1 Weight decay

The overfitting problem is more likely to occur with more expressive models. One might want to limit the expressive power of a model to reduce overfitting but without modifying too much the structure of the model (e.g. the shape of the scoring function of a KG embedding model). A simple method to do that is to add a penalty term to the loss function that depends only on the parameters of the model. Using the notation of Algorithm 9, the loss $\mathcal{L}(W)$ might become $\mathcal{L}(W) + \alpha\Omega(W)$ where α weights the contribution of the norm penalty Ω. The latter can have several forms. Two usual possibilities are the L^1 and L^2 regularization that simply consist in summing the L^1 or L^2 norms of the weights. Choosing a norm regularization is not insignificant as each type of regularization can have a specific impact on the geometry of the learned parameters. See Section 7.1 of [52] for more details.

A.2.2 Early-stopping

By design gradient-based optimization techniques aim at minimizing the training loss, e.g., the loss function evaluated on the training set. However, in case of overfitting, a model minimizing the training loss might not minimize the test loss.

Quite reliably, the dynamics of model training follow the following pattern: given a validation set distinct from the training one, both training and validation loss decrease at first but after a while, the validation loss starts increasing again while the training loss keeps decreasing. The moment of divergence marks the beginning of overfitting and the training procedure should be stopped at that moment to keep the parameter configuration that minimizes the validation loss used as a proxy to measure the generalization ability of the model. Early stopping is the algorithm that regularly measures the validation loss and keeps the parameters in memory to be able to stop training at the proper time. It is the topic of Section 7.8 of [52].

A.2.3 Dropout

Dropout increases the robustness of the model to noisy data and thus reduces overfitting simply by randomly omitting some weights of the network while training [113]. Intuitively, this forces the model to spread the computations on various neural paths in the network. This prevents a phenomenon, called complex co-adaptation, where some neurons are very highly dependent on the output of others. A small perturbation of the one can highly change the behavior of the other. It is detailed in Section 7.12 of [52].

Appendix B

Dirichlet problem on directed graphs

Let $G = (V, E)$ be a directed graph of n nodes and m edges. Let $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix of G and let $D = \text{diag}(A \cdot \mathbb{1}_n)$ be its out-degree matrix. D is supposed positive. $L = D - A$ denotes the graph Laplacian. Note that A and L are not necessarily symmetric because G is directed.

B.1 Definitions

Definition 29 (Sink)**.** *In a directed graph, a sink is a node with no outgoing edge. G has no sink if and only if the degree matrix D is positive.*

Definition 30 (Strong Connectedness)**.** *A directed graph is said to be strongly connected if any vertex is reachable from any other vertex.*

B.2 Dirichlet problem

Recall the Dirichlet problem as formulated in Chapter 6. It consists in solving the heat equation (6.2) in the presence of boundary conditions. Let S be some strict subset of V and assume that the temperature of each *seed* node $i \in S$ is set at some fixed value $T_i^{(S)}$. We are interested in the evolution of the temperatures of the other nodes. A synthetic formulation of the Dirichlet problem is the following:

$$\begin{cases} \frac{\partial \boldsymbol{T}}{\partial t} = -L\boldsymbol{T} \\ \forall t, \forall s \in S, \boldsymbol{T}_s(t) = \boldsymbol{T}_s(0) \end{cases} \tag{B.1}$$

We are interested in finding the vector of temperatures at equilibrium, i.e., such that $\forall i \notin S, (L\boldsymbol{T})_i = 0$ with the boundary conditions $\boldsymbol{T}^{(S)} \in \mathbb{R}^{|S|}$. Let $P = D^{-1}A$ be the transition matrix of a random walk on G, which is well defined because D is invertible because supposed positive. The previous equation can be re-written as

$$\forall i \notin S, \boldsymbol{T}_i = (P\boldsymbol{T})_i \tag{B.2}$$

That is the temperature of each node at equilibrium is the average temperature of its out-neighbors.

Without any loss of generality, let us assume that the boundary nodes S are indexed from $n - s$ to n where $s = |S|$. The vector of temperatures $\boldsymbol{T}$ can then be written $\boldsymbol{T} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{T}^{(S)} \end{pmatrix}$ with $\boldsymbol{x} \in \mathbb{R}^{n-s}$ the vector of unknown temperatures and $\boldsymbol{T}^{(S)} \in \mathbb{R}^s$ the

vector of known boundary temperatures. The transition matrix P can also be written in block form as $P = \begin{pmatrix} Q & R \\ \alpha & \beta \end{pmatrix}$. Now let us remark that (B.2) can be rewritten as in (B.3) in which only the blocks Q and R are involved.

$$\boldsymbol{x} = Q\boldsymbol{x} + R\boldsymbol{T}^{(S)} \tag{B.3}$$

B.3 Existence and unicity of the solutions

If there is a path from any node of $V \setminus S$ to at least one node in S, let us remark that we can assume without loss of generality that $\alpha = 0$ and $\beta = I_S$. This does not change equations (B.2) and (B.3). Intuitively, this is acceptable because out-neighbors of nodes in S have no impact on their temperatures (because it is set to $\boldsymbol{T}^{(S)}$) so the edges leaving S might as well be removed. Some self-loops are added to each node in S ($\beta = I_S$) in order to keep P stochastic. The form of P is then

$$P = \begin{pmatrix} Q & R \\ 0 & I_s \end{pmatrix} \tag{B.4}$$

Note that for the Markov chain associated to the transition matrix P, S are *absorbing states*: once entered they cannot be left. If there is a path from any node of $V \setminus S$ to at least one node in S, then S are the only ergodic states of the process and $V \setminus S$ are all transient states.

Lemma 1. *In any finite Markov chain, no matter where the process starts, the probability after n steps that the process is in an ergodic state tends to 1 as n tends to infinity.*

Proof. A proof of this lemma can be found in Chapter 3 of *Finite Markov Chains* by John G. Kemeny and J. Laurie Snell [64]. □

Proposition 2. *If there is a path from any node of G to at least one node in S, then* $\lim_{k\to\infty} Q^k = 0$.

Proof. Given $(i, j) \in \{1, \dots, n-s\}^2$ and $k \in \mathbb{N}$, $(Q^k)_{i,j}$ is by definition the probability that the Markov chain defined by the transition matrix P goes from node i to node j in k steps without leaving the nodes $\{1, \dots, n-s\}$, i.e., without entering any absorbing state in S. Lemma 1 tells us that the probability of entering those states goes to 1. This shows that $(Q^k)_{i,j} \to_{k\to\infty} 0$ for any i, j, concluding the proof. □

Proposition 3. *Let $A \in \mathbb{R}^{n\times n}$, if A^k tends to 0 as k tends to infinity then $(I - A)$ is invertible and $(I_n - A)^{-1} = \sum_{k=0}^{\infty} A^k$.*

Proof. A proof of this result can be found in Chapter 1 of *Finite Markov Chains* by John G. Kemeny and J. Laurie Snell [64]. □

If it is assumed that there is a path from any node of $V \setminus S$ to at least one node in S, applying Proposition 2 and Proposition 3 results in $(I_{n-s} - Q)$ being invertible. Eventually, (B.3) leads directly to the exact solution of the Dirichlet problem:

$$\boldsymbol{x}^* = (I_{n-s} - Q)^{-1} R\boldsymbol{T}^{(S)} \tag{B.5}$$

Algorithm 10: Discrete time algorithm.

Data: $P = \begin{pmatrix} Q & R \\ 0 & I_s \end{pmatrix}$ transition matrix of the random walk,
S set of seeds,
$\boldsymbol{T}^{(S)}$ temperatures of the seeds,
K number of iterations
Result: T vector of temperatures
$\boldsymbol{x} \leftarrow \text{mean}(\boldsymbol{T}^{(S)})$;
for $k = 1, .., K$ **do**
  $\boldsymbol{x} \leftarrow Q\boldsymbol{x} + R\boldsymbol{T}^{(S)}$;

B.4 Convergence in discrete time

Solving the Dirichlet problem in discrete time can be done using Algorithm 10. This section presents a proof of convergence.

Definition 31. *The spectral radius of a matrix A is defined as the largest absolute value of one of its eigenvalues. It is noted $\rho(A) \in \mathbb{R}^*$.*

Definition 32. *Let $\mathcal{N}$ a vector norm, $\mathcal{N}$ induces a matrix norm noted $|||\cdot|||_{\mathcal{N}}$ and defined as follows for any $A \in \mathbb{R}^{n\times n}$:*

$$|||A|||_{\mathcal{N}} = \sup_{\boldsymbol{x}\in\mathbb{R}^n : \mathcal{N}(\boldsymbol{x})=1} \mathcal{N}(A\boldsymbol{x})$$

For any vector norm $\mathcal{N}$, any $A \in \mathbb{R}^{n\times n}$ and any $\boldsymbol{x} \in \mathbb{R}^n$, $\mathcal{N}(A\boldsymbol{x}) \leq |||A|||_{\mathcal{N}} \cdot \mathcal{N}(\boldsymbol{x})$.

Lemma 2. *Given $Q \in \mathbb{R}^{n\times n}$, if $Q^k \to_{k\to\infty} 0$ then there is at least one vector norm $\mathcal{N}$ such that the induced matrix norm verifies $|||Q|||_{\mathcal{N}} < 1$.*

Proof. A proof of this lemma can be found in *Introduction à l'analyse numérique matricielle et à l'optimisation* by Philippe G. Ciarlet (Theorem 1.5-1). [29] □

Proposition 4. *If there is a path from any node of $V \setminus S$ to at least one node in S, then Algorithm 10 converges to the exact solution $\boldsymbol{x}^* = (I_{n-s} - Q)^{-1}R\boldsymbol{T}^{(S)}$.*

Proof. Let us assume that there is a path from any node of $V \setminus S$ to at least one node in S. Then $\boldsymbol{X}^* = (I_{n-s} - Q)^{-1}R\boldsymbol{T}^{(S)}$ is well defined and is the exact solution to the Dirichlet problem at hand (see Section B.3). Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be defined as:

$$\begin{aligned} f\colon \mathbb{R}^n &\to \mathbb{R}^n \\ \boldsymbol{x} &\mapsto Q\boldsymbol{x} + R\boldsymbol{T}^{(S)} \end{aligned}$$

Proving the convergence of Algorithm 10 comes down to proving that the iterated function sequence $\left(f^k(\boldsymbol{x}^{(0)})\right)_{k\in\mathbb{N}}$ converges to $\boldsymbol{x}^*$ given any initial state $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$. Indeed, the unknown temperature vector at time step $t \geq 1$ is $\boldsymbol{x}^{(t)} = f(\boldsymbol{x}^{(t-1)}) = f^t(\boldsymbol{x}^{(0)})$.

Applying Proposition 2 gives $Q^k \to_{k\to\infty} = 0$. Now let $\mathcal{N}$ be the vector norm provided by Lemma 2. Let $(\boldsymbol{x}, \boldsymbol{y}) \in (\mathbb{R}^n)^2$.

$$\begin{aligned} \mathcal{N}(f(\boldsymbol{x}) - f(\boldsymbol{y})) &= \mathcal{N}(Q\boldsymbol{x} - Q\boldsymbol{y}) \\ &= \mathcal{N}(Q(\boldsymbol{x} - \boldsymbol{y})) \\ &\leq |||Q|||_{\mathcal{N}} \cdot \mathcal{N}(\boldsymbol{x} - \boldsymbol{y}) \end{aligned} \tag{B.6}$$

Equation (B.6) shows that f is a contraction because $|||Q|||_{\mathcal{N}} < 1$ by definition of $\mathcal{N}$. Let us apply the Banach-Picard fixed point theorem [5] to the contraction f using

the completeness of $\mathbb{R}^n$. It yields that there exists a unique fixed point of f and that any sequence $\left(f^k(\boldsymbol{x}^{(0)})\right)_{k\in\mathbb{N}}$ converges to this point. $\boldsymbol{x}^*$ being a solution to $f(\boldsymbol{x}) = \boldsymbol{x}$, it is the unique fixed point and Algorithm 10 converges to it. □

Appendix C

The TF-IDF weighting scheme

In information retrieval, TF-IDF is a statistic that reflects how important a token is to a document given a collection of documents or corpus. It stands for Term Frequency and Inverse Document Frequency. First introduced by Sparck Jones in 1972 [112], it became vastly used in natural language processing to filter stop-words for example.

Let $\mathcal{W}$ be a set of words and $\mathcal{D} = \{d_1, \dots, d_n\}$ be a corpus of n documents. Suppose each document d_i is a sequence of words from $\mathcal{W}$.

Definition 33 (Term Frequency)**.** *Given a word $w \in \mathcal{W}$ and a document $d \in \mathcal{D}$, the term frequency $tf(w, d)$ is the relative frequency of term w in document d:*

$$tf(w, d) = \frac{f_{w,d}}{\sum_{\tilde{w} \in \mathcal{W}} f_{\tilde{w},d}}$$

$f_{w,d}$ is the raw count of the number of occurrences of word w in document d.

Intuitively, the larger the term frequency the more important a word must be. There are however some exceptions with words that are frequent in all the documents and that does not allow to discriminate them.

Definition 34 (Inverse Document Frequency)**.** *The inverse document frequency is a measure of how specific a word $w \in \mathcal{W}$ is to a given document $d \in \mathcal{D}$. It is defined as:*

$$idf(w, d) = \log \frac{n}{|\{d \in \mathcal{D} : w \in d\}|}$$

Intuitively, in a English corpus, stop words (e.g. *"the"*, *"it"*) are likely to have an *idf* close to 0 in all the documents of the corpus.

Definition 35 (TF-IDF)**.** *Given a word $w \in \mathcal{W}$ and a document $d \in \mathcal{D}$, the TF-IDF is computed as the product of the term frequency and the inverse document frequency:*

$$tdidf(w, d) = tf(w, d) \times idf(w, d) = \frac{f_{w,d}}{\sum_{\tilde{w} \in \mathcal{W}} f_{\tilde{w},d}} \times \log \frac{n}{|\{d \in \mathcal{D} : w \in d\}|}$$

There exist some variants to these definitions but the ones reported here are the most common. Extensive analysis and examples can be found in Chapter 3 of [107].

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Akrami, F., Saeef, M. S., Zhang, Q., Hu, W., and Li, C. Realistic re-evaluation of knowledge graph completion methods: An experimental study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 1995–2010.

[3] Alpaydin, E., and Kaynak, C. Optical recognition of handwritten digits data set.

[4] Balazevic, I., Allen, C., and Hospedales, T. TuckER: Tensor Factorization for Knowledge Graph Completion. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* (Hong Kong, China, Nov. 2019), Association for Computational Linguistics, pp. 5185–5194.

[5] Banach, S. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae 3*, 1 (1922), 133–181.

[6] Behler, J. Constructing high-dimensional neural network potentials: A tutorial review. *International Journal of Quantum Chemistry 115*, 16 (2015), 1032–1050. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.24890.

[7] Berberidis, D., Nikolakopoulos, A. N., and Giannakis, G. B. Adadif: Adaptive diffusions for efficient semi-supervised learning over graphs. In *2018 IEEE International Conference on Big Data (Big Data)* (2018), pp. 92–99.

[8] Bertinetto, L., Mueller, R., Tertikas, K., Samangooei, S., and Lord, N. A. Making Better Mistakes: Leveraging Class Hierarchies With Deep Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 12506–12515.

[9] Bishop, C. M. *Pattern Recognition And Machine Learning*, 1st ed. 2006. corr. 2nd printing 2011 édition ed. Springer-Verlag New York Inc., 2011.

[10] Biswas, R., Portisch, J., Paulheim, H., Sack, H., and Alam, M. Entity Type Prediction Leveraging Graph Walks and Entity Descriptions. In *The Semantic Web – ISWC 2022* (Cham, 2022), U. Sattler, A. Hogan, M. Keet, V. Presutti, J. P. A. Almeida, H. Takeda, P. Monnin, G. Pirrò, and C. d'Amato, Eds., Lecture Notes in Computer Science, Springer International Publishing, pp. 392–410.

[11] Biswas, R., Sofronova, R., Sack, H., and Alam, M. Cat2type: Wikipedia category embeddings for entity typing in knowledge graphs. In *Proceedings of the 11th on Knowledge Capture Conference* (New York, NY, USA, 2021), K-CAP '21, Association for Computing Machinery, p. 81–88.

[12] Bizer, C., Heath, T., Idehen, K., and Berners-Lee, T. Linked data on the Web. In *WWW* (2008).

[13] Blake, C. Wine recognition data.

[14] Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, Association for Computing Machinery, p. 1247–1250.

[15] Bonald, T. Lecture notes on spectral embedding of graphs, Jan. 2019.

[16] Bordes, A., Glorot, X., Weston, J., and Bengio, Y. A Semantic Matching Energy Function for Learning with Multi-relational Data. *Machine Learning 94*, 2 (Feb. 2014), 233–259.

[17] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 2787–2795.

[18] Boschin, A. TorchKGE: Knowledge Graph Embedding in Python and PyTorch. *KDD-IWKG 2020* (Aug. 2020), 6.

[19] Boschin, A., and Bonald, T. WikiDataSets : Standardized sub-graphs from Wikidata. *arXiv:1906.04536 [cs, stat]* (June 2019). arXiv: 1906.04536.

[20] Boschin, A., and Bonald, T. Enriching wikidata with semantified wikipedia hyperlinks. In *Proceedings of the 2nd Wikidata Workshop co-located with the 20th International Semantic Web Conference* (Oct 2021).

[21] Boschin, A., Bonald, T., and Jeanmougin, M. A self-encoder for learning nearest neighbors. *Preprint*, 2023.

[22] Boschin, A., Jain, N., Keretchashvili, G., and Suchanek, F. M. Combining embeddings and rules for fact prediction. In *Proceedings of the 15th Reasoning Web Summer School* (Bolzano, Italy, September 2019).

[23] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems* (2020), vol. 33, Curran Associates, Inc., pp. 1877–1901.

[24] Bruna, J., Zaremba, W., Szlam, A., and Lecun, Y. Spectral networks and locally connected networks on graphs. *International Conference on Learning Representations (ICLR2014)* (Apr. 2014).

[25] Brust, C.-A., and Denzler, J. Integrating Domain Knowledge: Using Hierarchies to Improve Deep Classifiers. In *Pattern Recognition: 5th Asian Conference, ACPR 2019, Auckland, New Zealand, November 26–29, 2019, Revised Selected Papers, Part I* (Berlin, Heidelberg, Nov. 2019), Springer-Verlag, pp. 3–16.

[26] Cai, L., and Wang, W. Y. KBGAN: Adversarial Learning for Knowledge Graph Embeddings. *arXiv:1711.04071 [cs]* (Nov. 2017). arXiv: 1711.04071.

[27] Chen, D., Lin, Y., Li, W., Li, P., Zhou, J., and Sun, X. Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View. *Proceedings of the AAAI Conference on Artificial Intelligence 34*, 04 (Apr. 2020), 3438–3445. Number: 04.

[28] Chung, F. R. K. *Spectral Graph Theory*. American Mathematical Society, 1997.

[29] Ciarlet, P. G. *Introduction à l'analyse numérique matricielle et à l'optimisation*. Dunod, 1998. Google-Books-ID: L1TMPQAACAAJ.

[30] Cortes, C., and Vapnik, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[31] Costabello, L., Pai, S., Van, C. L., McGrath, R., and McCarthy, N. AmpliGraph: a Library for Representation Learning on Knowledge Graphs, Mar. 2019.

[32] Cover, T., and Hart, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory 13*, 1 (Jan. 1967), 21–27. Conference Name: IEEE Transactions on Information Theory.

[33] de Lara, N., and Bonald, T. A Consistent Diffusion-Based Algorithm for Semi-Supervised Classification on Graphs. *arXiv:2008.11944 [cs]* (Aug. 2020). arXiv: 2008.11944.

[34] Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems* (2016), vol. 29, Curran Associates, Inc.

[35] Dettmers, T., Minervini, P., Stenetorp, P., and Riedel, S. Convolutional 2d Knowledge Graph Embeddings. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence* (New Orleans, LA, USA, Feb. 2018), vol. 32, AAAI Press.

[36] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (Minneapolis, Minnesota, June 2019), Association for Computational Linguistics, pp. 4171–4186.

[37] Diderot, D., and Le Rond d'Alember, J. *Encyclopédie ou Dictionnaire raisonné des sciences, des arts et des métiers*. Briasson, 1753.

[38] Ding, B., Wang, Q., Wang, B., and Guo, L. Improving Knowledge Graph Embedding Using Simple Constraints. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Melbourne, Australia, July 2018), Association for Computational Linguistics, pp. 110–121.

[39] Dong, X. L., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., and Zhang, W. Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, Aug. 2014), pp. 601–610.

[40] d'Amato, C., Quatraro, N. F., and Fanizzi, N. Injecting Background Knowledge into Embedding Models for Predictive Tasks on Knowledge Graphs. In *The Semantic Web* (Cham, 2021), R. Verborgh, K. Hose, H. Paulheim, P.-A. Champin, M. Maleshkova, O. Corcho, P. Ristoski, and M. Alam, Eds., Lecture Notes in Computer Science, Springer International Publishing, pp. 441–457.

[41] Ebisu, T., and Ichise, R. TorusE: Knowledge Graph Embedding on a Lie Group. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence* (New Orleans, LA, USA, Feb. 2018), AAAI Press, pp. 1819–1826.

[42] Etzioni, O., Kok, S., Soderland, S., Cafarella, M., Popescu, A.-M., Weld, D. S., Downey, D., Shaked, T., and Yates, A. Web-Scale Information Extraction in KnowItAll (Preliminary Results). In *Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, May 2004), pp. 100–110.

[43] Fatemi, B., Ravanbakhsh, S., and Poole, D. Improved knowledge graph embedding using background taxonomic information. In *AAAI* (2019), vol. 33.

[44] Fellbaum, C. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[45] Ferré, S. Link Prediction in Knowledge Graphs with Concepts of Nearest Neighbours. In *The Semantic Web* (Cham, 2019), P. Hitzler, M. Fernández, K. Janowicz, A. Zaveri, A. J. Gray, V. Lopez, A. Haller, and K. Hammar, Eds., Lecture Notes in Computer Science, Springer International Publishing, pp. 84–100.

[46] Fisher, R. A. The use of multiple measurements in taxonomic problems. *Annals of Eugenics 7*, 2 (1936), 179–188.

[47] Galárraga, L., Symeonidou, D., and Moissinac, J.-C. Rule Mining for Semantifying Wikilinks. In *LDOW@WWW* (2015).

[48] Galárraga, L., Teflioudi, C., Hose, K., and Suchanek, F. M. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. In *VLDBJ* (2015).

[49] Galárraga, L. A., Teflioudi, C., Hose, K., and Suchanek, F. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22nd international conference on World Wide Web* (Rio de Janeiro, Brazil, 2013), ACM Press, pp. 413–422.

[50] Gammerman, A., Vovk, V., and Vapnik, V. Learning by transduction. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence* (San Francisco, CA, USA, 1998), UAI'98, Morgan Kaufmann Publishers Inc., p. 148–155.

[51] German, B. Glass identification database.

[52] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[53] Guo, S., Wang, Q., Wang, L., Wang, B., and Guo, L. Jointly embedding knowledge graphs and logical rules. In *EMNLP* (2016).

[54] Guo, S., Wang, Q., Wang, L., Wang, B., and Guo, L. Knowledge graph embedding with iterative guidance from soft rules. In *AAAI* (2018).

[55] Han, X., Cao, S., Lv, X., Lin, Y., Liu, Z., Sun, M., and Li, J. OpenKE: An Open Toolkit for Knowledge Embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Brussels, Belgium, 2018), Association for Computational Linguistics, pp. 139–144.

[56] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[57] Hitchcock, F. L. The Expression of a Tensor or a Polyadic as a Sum of Products. *Journal of Mathematics and Physics 6*, 1-4 (1927), 164–189. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sapm192761164.

[58] Hofmann, H. Statlog (german credit data) data set.

[59] Jain, N., Tran, T.-K., Gad-Elrab, M. H., and Stepanova, D. Improving Knowledge Graph Embeddings with Ontological Reasoning. In *The Semantic Web – ISWC 2021* (Cham, 2021), A. Hotho, E. Blomqvist, S. Dietze, A. Fokoue, Y. Ding, P. Barnaghi, A. Haller, M. Dragoni, and H. Alani, Eds., Lecture Notes in Computer Science, Springer International Publishing, pp. 410–426.

[60] Ji, G., He, S., Xu, L., Liu, K., and Zhao, J. Knowledge Graph Embedding via Dynamic Mapping Matrix. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)* (Beijing, China, July 2015), Association for Computational Linguistics, pp. 687–696.

[61] Jin, H., Hou, L., Li, J., and Dong, T. Attributed and predictive entity embedding for fine-grained entity typing in knowledge bases. In *Proceedings of the 27th International Conference on Computational Linguistics* (Santa Fe, New Mexico, USA, Aug. 2018), Association for Computational Linguistics, pp. 282–292.

[62] Jin, H., Hou, L., Li, J., and Dong, T. Fine-grained entity typing via hierarchical multi graph convolutional networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* (Hong Kong, China, Nov. 2019), Association for Computational Linguistics, pp. 4969–4978.

[63] Kadlec, R., Bajgar, O., and Kleindienst, J. Knowledge Base Completion: Baselines Strike Back. *arXiv:1705.10744 [cs]* (May 2017). arXiv: 1705.10744.

[64] Kemeny, J. G., and Snell, J. L. *Finite Markov Chains*. Princeton University Press, 1960.

[65] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[66] Kingma, D. P., and Welling, M. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*. (Dec. 2013).

[67] Kipf, T. N., and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations (ICLR)* (Toulon, France, Apr. 2017), OpenReview.net.

[68] Kolda, T. G., and Bader, B. W. Tensor Decompositions and Applications. *SIAM Review* (2009).

[69] Kosmopoulos, A., Partalas, I., Gaussier, E., Paliouras, G., and Androutsopoulos, I. Evaluation measures for hierarchical classification: a unified view and novel approaches. *Data Mining and Knowledge Discovery 29*, 3 (May 2015), 820–865.

[70] Kotnis, B., and Nastase, V. Analysis of the impact of negative sampling on link prediction in knowledge graphs. *arXiv preprint arXiv:1708.06816* (2017).

[71] Krompass, D., Baier, S., and Tresp, V. Type-constrained representation learning in knowledge graphs. In *The Semantic Web - ISWC 2015* (Cham, 2015), M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, K. Thirunarayan, and S. Staab, Eds., Springer International Publishing, pp. 640–655.

[72] Lacroix, T., Usunier, N., and Obozinski, G. Canonical tensor decomposition for knowledge base completion. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 2863–2872.

[73] Lajus, J., Galárraga, L., and Suchanek, F. M. Fast and Exact Rule Mining with AMIE 3. In *ESWC* (2020).

[74] Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., and Bizer, C. DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web 6* (2015), 167–195.

[75] Lengyel, B. A., and Stone, M. H. Elementary Proof of the Spectral Theorem. *Annals of Mathematics 37*, 4 (1936), 853–864.

[76] Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of the 2nd Conference on Systems and Machine Learning* (Palo Alto, CA, USA, Mar. 2019).

[77] Li, Q., An, S., Liu, W., and Li, L. Semisupervised learning on graphs with an alternating diffusion process. *IEEE Transactions on Neural Networks and Learning Systems 32*, 7 (2021), 2862–2874.

[78] Lin, Y., Liu, Z., Sun, M., Liu, Y., and Zhu, X. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, TX, USA, Feb. 2015), AAAI Press, pp. 2181–2187.

[79] Liu, H., and Singh, P. ConceptNet — A Practical Commonsense Reasoning Tool-Kit. *BT Technology Journal 22*, 4 (Oct. 2004), 211–226.

[80] Liu, H., Wu, Y., and Yang, Y. Analogical Inference for Multi-Relational Embeddings. In *Proceedings of the 34th International Conference on Machine Learning* (Sydney, Australia, May 2017), vol. 70, PMLR, pp. 2168–2178.

[81] Ltd., B. M. R. Bupa liver disorders.

[82] Lucena, B. Loss Functions for Classification using Structured Entropy, June 2022. Publication Title: arXiv e-prints ADS Bibcode: 2022arXiv220607122L Type: article.

[83] Mahdisoltani, F., Biega, J., and Suchanek, F. M. YAGO3: A Knowledge Base from Multilingual Wikipedias. In *CIDR* (Jan. 2013).

[84] Mangasarian, . L., and Wolberg, W. H. Cancer diagnosis via linear programming. *SIAM News 23*, 5 (9 1990), 1–18.

[85] McDermott, D. Building large knowledge-based systems: Representation and inference in the cyc project: D.B. Lenat and R.V. Guha. *Artificial Intelligence 61*, 1 (May 1993), 53–63.

[86] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems* (Lake Tahoe, NV, USA, Dec. 2013), vol. 2, Curran Associates Inc., pp. 3111–3119.

[87] Minervini, P., Costabello, L., Muñoz, E., Nováček, V., and Vandenbussche, P.-Y. Regularizing knowledge graph embeddings via equivalence and inversion axioms. In *ECML PKDD* (2017).

[88] Mitchell, T. Conditions for the Equivalence of Hierarchical and Non Hierarchical Bayesian Classifiers. Tech. rep., Carnegie Mellon University, 1998.

[89] Nakai, K., and Kanehisa, M. A knowledge base for predicting protein localization sites in eukaryotic cells. *Genomics 14* (1992), 897–911.

[90] Nassiri, A., Pernelle, N., Saïs, F., and Quercini, G. Knowledge Graph Refinement based on Triplet BERT-Networks. *Presented at the DeepOntoNLP Workshop co-located with ESWC 2022* (May 2022).

[91] Nathani, D., Chauhan, J., Sharma, C., and Kaul, M. Learning attention-based embeddings for relation prediction in knowledge graphs. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (2019).

[92] Navigli, R., and Ponzetto, S. P. BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence 193* (Dec. 2012), 217–250.

[93] Nguyen, D. Q., Nguyen, T. D., Nguyen, D. Q., and Phung, D. A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2018), vol. 2, pp. 327–333.

[94] Nickel, M., Rosasco, L., and Poggio, T. Holographic Embeddings of Knowledge Graphs. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (Phoenix, AZ, USA, Feb. 2016), AAAI Press, pp. 1955–1961. arXiv: 1510.04935.

[95] Nickel, M., Tresp, V., and Kriegel, H.-P. A Three-way Model for Collective Learning on Multi-relational Data. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (Bellevue, WA, USA, 2011), ICML'11, Omnipress, pp. 809–816.

[96] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue 6*, 2 (Mar. 2008), 40–53.

[97] Ortona, S., Meduri, V. V., and Papotti, P. Robust discovery of positive and negative rules in knowledge bases. In *ICDE* (2018).

[98] Paris, P.-H., and Suchanek, F. Non-named Entities – The Silent Majority. In *The Semantic Web: ESWC 2021 Satellite Events* (Cham, 2021), R. Verborgh, A. Dimou, A. Hogan, C. d'Amato, I. Tiddi, A. Bröring, S. Mayer, F. Ongenae, R. Tommasini, and M. Alam, Eds., Springer International Publishing, pp. 131–135.

[99] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[100] Pearson, K. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2*, 11 (1901), 559–572.

[101] Pellissier Tanon, T., Weikum, G., and Suchanek, F. YAGO 4: A Reasonable Knowledge Base. In *The Semantic Web* (Cham, 2020), A. Harth, S. Kirrane, A.-C. Ngonga Ngomo, H. Paulheim, A. Rula, A. L. Gentile, P. Haase, and M. Cochez, Eds., Lecture Notes in Computer Science, Springer International Publishing, pp. 583–596.

[102] Peng, N., Poon, H., Quirk, C., Toutanova, K., and Yih, W.-t. Cross-sentence n-ary relation extraction with graph LSTMs. *Transactions of the Association for Computational Linguistics 5* (2017), 101–115.

[103] Razniewski, S., Suchanek, F. M., and Nutt, W. But what do we actually know? In *AKBC workshop* (2016).

[104] Rossi, A., Barbosa, D., Firmani, D., Matinata, A., and Merialdo, P. Knowledge graph embedding for link prediction: A comparative analysis. *ACM Trans. Knowl. Discov. Data 15*, 2 (jan 2021).

[105] Ruffinelli, D., Broscheit, S., and Gemulla, R. You can teach an old dog new tricks! on training knowledge graph embeddings. In *International Conference on Learning Representations* (2020).

[106] Sakor, A., Singh, K., Patel, A., and Vidal, M.-E. Falcon 2.0: An entity and relation linking tool over wikidata. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2020), CIKM '20, Association for Computing Machinery, p. 3141–3148.

[107] Salton, G., and McGill, M. J. *Introduction to modern information retrieval*. New York : McGraw-Hill, 1983.

[108] Schlichtkrull, M., Kipf, T. N., Bloem, P., Berg, R. v. d., Titov, I., and Welling, M. Modeling Relational Data with Graph Convolutional Networks. In *Proceeding of the 15th European Semantic Web Conference* (Heraklion, Crete, Greece, June 2018), pp. 593–607.

[109] Sigillito, V. Johns hopkins university ionosphere database.

[110] Silla, C. N., and Freitas, A. A. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery 22*, 1 (Jan. 2011), 31–72.

[111] Socher, R., Chen, D., Manning, C. D., and Ng, A. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 926–934.

[112] Sparck Jones, K. A Statistical Interpretation of Term Specificity and its Application in Retrieval. *Journal of Documentation 28*, 1 (Jan. 1972), 11–21. Publisher: MCB UP Ltd.

[113] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research 15*, 56 (2014), 1929–1958.

[114] Suchanek, F. M., Abiteboul, S., and Senellart, P. Paris: Probabilistic alignment of relations, instances, and schema. In *VLDB* (2012).

[115] Suchanek, F. M., Kasneci, G., and Weikum, G. YAGO: A Core of Semantic Knowledge Unifying WordNet and Wikipedia. In *Proceedings of the 16th international conference on World Wide Web* (Banff, Alberta, Canada, May 2007), ACM, pp. 697–706.

[116] Suchanek, F. M., Lajus, J., Boschin, A., and Weikum, G. Knowledge representation and rule mining in entity-centric knowledge bases. In *Reasoning Web. Explainable Artificial Intelligence - 15th International Summer School 2019, Bolzano, Italy, September 20-24, 2019* (2019), Springer, pp. 110–152.

[117] Sun, Z., Vashishth, S., Sanyal, S., Talukdar, P., and Yang, Y. A Re-evaluation of Knowledge Graph Completion Methods. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (Online, July 2020), Association for Computational Linguistics, pp. 5516–5522.

[118] Thanou, D., Dong, X., Kressner, D., and Frossard, P. Learning heat diffusion graphs. *IEEE Transactions on Signal and Information Processing over Networks 3*, 3 (2017), 484–499.

[119] Toutanova, K., Chen, D., Pantel, P., Poon, H., Choudhury, P., and Gamon, M. Representing Text for Joint Embedding of Text and Knowledge

Bases. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (Lisbon, Portugal, 2015), Association for Computational Linguistics, pp. 1499–1509.

[120] Trouillon, T. *Complex-Valued Embedding Models for Knowledge Graphs*. Phd thesis, Université Grenoble Alpes, Sept. 2017.

[121] Trouillon, T., and Nickel, M. Complex and Holographic Embeddings of Knowledge Graphs: A Comparison. *arXiv:1707.01475 [cs, stat]* (July 2017). arXiv: 1707.01475.

[122] Trouillon, T., Welbl, J., Riedel, S., Gaussier, E., and Bouchard, G. Complex Embeddings for Simple Link Prediction. In *Proceedings of the 33rd International Conference on Machine Learning* (New York, NY, USA, June 2016), vol. 48, pp. 2071–2080.

[123] Tucker, L. R. Some mathematical notes on three-mode factor analysis. *Psychometrika 31*, 3 (Sept. 1966), 279–311.

[124] van Rossum, G., Warsaw, B., and Coghlan, N. Style guide for Python code. PEP 8, Python Software Foundation, 2001.

[125] Vashishth, S., Sanyal, S., Nitin, V., and Talukdar, P. Composition-based multi-relational graph convolutional networks. In *ICLR* (2019).

[126] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *The Journal of Machine Learning Research 11* (Dec. 2010), 3371–3408.

[127] Vrandečić, D., and Krötzsch, M. Wikidata: A Free Collaborative Knowledge Base. *Communications of the ACM 57* (2014), 78–85.

[128] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[129] Wang, L., Cao, Z., de Melo, G., and Liu, Z. Relation Classification via Multi-Level Attention CNNs. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Berlin, Germany, Aug. 2016), Association for Computational Linguistics, pp. 1298–1307.

[130] Wang, R., Li, B., Hu, S., Du, W., and Zhang, M. Knowledge Graph Embedding via Graph Attenuated Attention Networks. *IEEE Access 8* (2020), 5212–5224. Conference Name: IEEE Access.

[131] Wang, Z., Zhang, J., Feng, J., and Chen, Z. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence* (2014), AAAI'14, AAAI Press, p. 1112–1119.

[132] Weller, T., and Acosta, M. Predicting instance type assertions in knowledge graphs using stochastic neural networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (New York, NY, USA, 2021), CIKM '21, Association for Computing Machinery, p. 2111–2118.

[133] Wilcke, W. X., Bloem, P., de Boer, V., van 't Veer, R., and van Harmelen, F. End-to-end entity classification on multimodal knowledge graphs. *ArXiv abs/2003.12383* (2020).

[134] Word Wide Web Consortium. RDF Primer, 2004.

[135] Word Wide Web Consortium. SPARQL 1.1 Query Language, 2013.

[136] Wu, C., Tygert, M., and LeCun, Y. A hierarchical loss and its problems when classifying non-hierarchically, 2019.

[137] Xie, R., Liu, Z., and Sun, M. Representation learning of knowledge graphs with hierarchical types. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (2016), IJCAI'16, AAAI Press, p. 2965–2971.

[138] Yaghoobzadeh, Y., and Schütze, H. Corpus-level fine-grained entity typing using contextual information. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (Lisbon, Portugal, Sept. 2015), Association for Computational Linguistics, pp. 715–725.

[139] Yaghoobzadeh, Y., and Schütze, H. Multi-level representations for fine-grained typing of knowledge base entities. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers* (Valencia, Spain, Apr. 2017), Association for Computational Linguistics, pp. 578–589.

[140] Yamada, I., Asai, A., Shindo, H., Takeda, H., and Matsumoto, Y. LUKE: Deep contextualized entity representations with entity-aware self-attention. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Online, Nov. 2020), Association for Computational Linguistics, pp. 6442–6454.

[141] Yang, B., Yih, W.-t., He, X., Gao, J., and Deng, L. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *Proceedings of the International Conference on Learning Representation* (Banff, Canada, Dec. 2014).

[142] Yang, X., Ren, S., Li, Y., Shen, K., Li, Z., and Wang, G. Relation linking for wikidata using bag of distribution representation. In *Natural Language Processing and Chinese Computing* (Cham, 2018), X. Huang, J. Jiang, D. Zhao, Y. Feng, and Y. Hong, Eds., Springer International Publishing, pp. 652–661.

[143] Ye, R., Li, X., Fang, Y., Zang, H., and Wang, M. A vectorized relational graph convolutional network for multi-relational network alignment. In *IJCAI* (2019).

[144] Yu, S. Y., Rokka Chhetri, S., Canedo, A., Goyal, P., and Faruque, M. A. A. Pykg2vec: A python library for knowledge graph embedding. *arXiv preprint arXiv:1906.04239* (2019).

[145] Zachary, W. W. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research 33*, 4 (1977), 452–473.

[146] Zhang, Y., Qi, P., and Manning, C. D. Graph convolution over pruned dependency trees improves relation extraction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (Brussels, Belgium, Oct.-Nov. 2018), Association for Computational Linguistics, pp. 2205–2215.

[147] Zhang, Y., Yao, Q., and Chen, L. Interstellar: searching recurrent architecture for knowledge graph embedding. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, Dec. 2020), NIPS'20, Curran Associates Inc., pp. 10030–10040.

[148] Zhang, Y., Yao, Q., Shao, Y., and Chen, L. NSCaching: Simple and Efficient Negative Sampling for Knowledge Graph Embedding. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (Apr. 2019), pp. 614–625. ISSN: 2375-026X.

[149] Zhuo, J., Zhu, Q., Yue, Y., Zhao, Y., and Han, W. A neighborhood-attention fine-grained entity typing for knowledge graph completion. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2022), WSDM '22, Association for Computing Machinery, p. 1525–1533.

ECOLE
DOCTORALE

Titre : Méthodes d'apprentissage automatique pour la complétion de graphes de connaissances

Mots clés : graphe de connaissances ; apprentissage automatique ; plongement ; graphe ; classification

Résumé : Un graphe de connaissances est un graphe orienté dont les nœuds sont des entités et les arêtes, typées par une relation, représentent des faits connus liant les entités. Ces graphes sont capables d'encoder une grande variété d'information mais leur construction et leur exploitation peut se révéler complexe. Historiquement, des méthodes symboliques ont permis d'extraire des règles d'interaction entre entités et relations, afin de corriger des anomalies ou de prédire des faits manquants. Plus récemment, des méthodes d'apprentissage de représentations vectorielles, ou plongements, ont tenté de résoudre ces mêmes tâches. Initialement purement algébriques ou géométriques, ces méthodes se sont complexifiées avec les réseaux de neurones profonds et ont parfois été combinées à des techniques symboliques antérieures.

Dans cette thèse, on s'intéresse tout d'abord au problème de l'implémentation. En effet, la grande diversité des bibliothèques utilisées rend difficile la comparaison des résultats obtenus par différents modèles. Dans ce contexte, la bibliothèque Python TorchKGE a été développée afin de proposer un environnement unique pour l'implémentation de modèles de plongement et un module hautement efficace d'évaluation par prédiction de liens. Cette bibliothèque repose sur l'accélération graphique de calculs tensoriels proposée par PyTorch, est compatible avec les bibliothèques d'optimisation usuelles et est disponible en source ouverte.

Ensuite, les travaux portent sur l'enrichissement automatique de Wikidata par typage des hyperliens liant les articles de Wikipedia. Une étude préliminaire a montré que le graphe des articles de Wikipedia est beaucoup plus dense que le graphe de connaissances correspondant dans Wikidata. Une nouvelle méthode d'entrainement impliquant les relations et une méthode d'inférence utilisant les types des entités ont été proposées et des expériences ont montré la pertinence de l'approche, y compris sur un nouveau jeu de données.

Enfin, le typage automatique d'entités est exploré comme une tâche de classification hiérarchique. Ceci a mené à la conception d'une fonction d'erreur hiérarchique, utilisée pour l'entrainement de modèles tensoriels, ainsi qu'un nouveau type d'encodeur. Des expériences ont permis une bonne compréhension de l'impact que peut avoir une connaissance a priori de la taxonomie des classes sur la classification. Elles ont aussi renforcé l'intuition que la hiérarchie peut être apprise à partir des données si le jeu est suffisamment riche.

Title : Machine learning techniques for automatic knowledge graph completion

Keywords : knowledge graph ; machine learning ; embedding ; graph ; classification

Abstract : A knowledge graph is a directed graph in which nodes are entities and edges, typed by a relation, represent known facts linking two entities. These graphs can encode a wide variety of information, but their construction and exploitation can be complex. Historically, symbolic methods have been used to extract rules about entities and relations, to correct anomalies or to predict missing facts. More recently, techniques of representation learning, or embeddings, have attempted to solve these same tasks. Initially purely algebraic or geometric, these methods have become more complex with deep neural networks and have sometimes been combined with pre-existing symbolic techniques.

In this thesis, we first focus on the problem of implementation. Indeed, the diversity of libraries used makes the comparison of results obtained by different models a complex task. In this context, the Python library TorchKGE was developed to provide a unique setup for the implementation of embedding models and a highly efficient inference evaluation module. This library relies on graphic acceleration of tensor computation provided by PyTorch, is compatible with widespread optimization libraries and is available as open source.

We then consider the automatic enrichment of Wikidata by typing the hyperlinks linking Wikipedia pages. A preliminary study showed that the graph of Wikipedia articles is much denser than the corresponding knowledge graph in Wikidata. A new training method involving relations and an inference method using entity types were proposed and experiments showed the relevance of the combined approach, including on a new dataset.

Finally, we explore automatic entity typing as a hierarchical classification task. That led to the design of a new hierarchical loss used to train tensor-based models along with a new type of encoder. Experiments on two datasets have allowed a good understanding of the impact a prior knowledge of class taxonomy can have on a classifier but also reinforced the intuition that the hierarchy can be learned from the features if the dataset is large enough.

Institut Polytechnique de Paris
91120 Palaiseau, France